MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 11

Institut d'Investigació
en Intel·ligència Artificial

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1	J. Puyol, *MILORD II: A Language for Knowledge–Based Systems*
Num. 2	J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*
Num. 3	Ll. Vila, *On Temporal Representation and Reasoning in Knowledge–Based Systems*
Num. 4	M. Domingo, *An Expert System Architecture for Identification in Biology*
Num. 5	E. Armengol, *A Framework for Integrating Learning and Problem Solving*
Num. 6	J.Ll. Arcos, *The Noos Representation Language*
Num. 7	J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*
Num. 8	P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*
Num. 9	F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*
Num. 10	M. Schorlemmer, *On Specifying and Reasoning with Special Relations*
Num. 11	M. López-Sánchez, *Approaches to Map Generation by means of Collaborative Autonomous Robots*

Approaches to Map Generation by means of Collaborative Autonomous Robots

Maite López-Sánchez

Foreword by Ramon López de Màntaras
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Ramon López de Màntaras
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Maite López-Sánchez
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

ISBN: 84-00-07835-7
Dip. Legal: B-47690-2000

Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Foreword

To the best of my knowledge, the work described in this monography is the first to have addressed the problem of map generation by means of a multi-robot approach. This has been also recognized by leading researchers in the field. Previous approaches had characterized the map generation problem only in terms of how the map is represented. In this work, a much better and complete characterization is proposed. This characterization takes into account not only the map representation (grid-based versus feature-based) but also the type of terrain to be mapped, the existing knowledge about the environment, the type of exploration and number of robots, the type of sensors used, and the use of the represented map.

The reader will also find original contributions in the three approaches to map generation described. The first two deal with indoor unknown structured environments and are based, respectively, on fuzzy sets and possibility theory techniques. The concept of possibility grid used in the second approach, is an original contribution that has very interesting advantages over probability grids both computationally (allows local computations) and conceptually (easy and natural representation of ignorance and modeling not only the occupied but also the free space). The first approach has been tested on real robots and the second one on simulated robots implementing a behavior-based architecture. The third approach involves an heterogeneous group of robots (airborne and ground) that collaborate in the mapping of an outdoor environment by grouping overlapping polygons representing obstacles. The results have been obtained also with real robots.

In this work, the reader will also find out how the robots communicate, what do they communicate, and how the obtained maps are used for planning, as well as detailed descriptions of the architecture of the robots, the simulator, the implementation, and the experiments performed.

As advisor of this work, my collaboration with the author has been very fruitful and enjoyable. I hope the reader will appreciate, and also enjoy, reading it.

Bellaterra, September 1999

Ramon López de Màntaras, IIIA (CSIC)

Contents

List of Figures

List of Tables

Abstract

The research of this thesis is focused on the map generation problem. It has been studied trough a compilation of three new and alternative multi-robot approaches. Their specification and implementation has provided us with the experience necessary to characterise the mapping problem based on starting assumptions and problem settings (i.e., type of environment, number of robots, gathered information, etc.).

Mapping approaches involve three phases: The map acquisition phase, which has been faced through the use of several robots that explore the environment. The map building phase, that represents and updates the gathered information as well as its associated uncertainty. And, the map processing phase, which depends on the subsequent use of the resulting map. Our proposed approaches face different instances of the mapping problem by considering:

- Unknown indoor and outdoor environments.
- Behaviour-based homogeneous and heterogeneous robots that explore the environment in the acquisition phase.
- Feature-based maps and grid map representations.
- Fuzzy sets, Possibility/Necessity values, and weighted costs for uncertainty treatment.
- Path planning and map completion in the map processing phase.

Resumen

La investigación llevada a cabo durante la realización de la presente tesis doctoral se centra en el problema de la generación de mapas. Dicho estudio ha sido realizado a través de la especificación e implementación de tres aproximaciones multirobot alternativas. La experiencia adquirida tanto al tratar el mismo problema con diferentes técnicas así como al considerar diversas disposiciones del problema (tales como el tipo de entorno a considerar, el número de robots a utilizar o la representación idónea de la información obtenida), nos ha permitido adquirir la abstracción necesaria para proponer una caracterización de las diferentes soluciones que han sido históricamente propuestas.

La generación de mapas consta de tres etapas principales: la adquisición de la información, la creación del mapa y su posterior tratamiento. La adquisición de la información es llevada a cabo por un grupo de robots que exploran un entorno desconocido. La creación del mapa se refiere a la etapa intermedia que conlleva la representación y el mantenimiento de la información. Finalmente, el postproceso del mapa corresponde al tratamiento del mapa necesario para su posterior uso. Las tres aproximaciones propuestas tratan diferentes instancias del problema de generación de mapas y consideran los siguientes aspectos:

- Entornos desconocidos de interiores y exteriores.
- Robots (tanto homogéneos como heterogéneos) que exploran el entorno guiados por una estrategia basada en comportamientos.
- Representaciones de mapas basadas en características y en discretizaciones del entorno.
- Tratamiento de la incertidumbre mediante conjuntos difusos, costes ponderados así como de valores de Posibilidad y Necesidad.
- Postproceso que consiste en la planificación de caminos y completado de mapas.

Resum

La recerca portada a terme durant la realització de la present tesi doctoral es centra en el problema de la generació de mapes. Aquest estudi ha estat realitzat a través de l'especificació i implementació de tres aproximacions multirobot alternatives. L'experiència adquirida tant al tractar el mateix problema amb diferents tècniques com al considerar diverses disposicions del problema (tal com el tipus d'entorn a considerar, el nombre de robots a emprar o la representació més adient de la informació obtinguda), ens ha permès adquirir l'abstracció necessària per a proposar una caracterització de les diferents solucions que han estat històricament proposades.

La generació de mapes consta de tres etapes principals: l'adquisició de la informació, la creació del mapa i el seu posterior tractament. L'adquisició de la informació es realitza mitjançant un grup de robots que exploren un entorn desconegut. La creació del mapa correspon a l'etapa intermitja que comporta la representació i el manteniment de la informació. Finalment, el postprocés del mapa es refereix al tractament necessari per al seu posterior ús. Les tres aproximacions proposades tracten diferents instàncies del problema de generació de mapes i consideren els següents aspectes:

- Entorns desconeguts d'interiors i exteriors.
- Robots (tant homogenis com heterogenis) que exploren l'entorn guiats per una estratègia basada en comportaments.
- Representacions de mapes basades en característiques i en discretitzacions de l'entorn.
- Tractament de la incertesa mitjançant conjunts difusos, costs ponderats així com de valors de Possibilitat i Necessitat.
- Postprocés que consisteix en la planificació de camins i completació de mapes.

Acknowledgements

This thesis has been mainly supported by the Spanish Council for Scientific Research (CSIC) and the Artificial Intelligence Research Institute (IIIA). Additional support has been provided by the Catalan "Comissionat per a Universitats i Recerca" (CUR), the Catalan "Centre de Referència en Tècniques Avançades de Producció" (CeRTAP), and a Spanish CICYT project TAP97-1209.

First and foremost, I would like to thank Ramon López de Màntaras, my Ph.D. advisor. In addition to giving me the opportunity of doing this thesis, he introduced me to the scientific 'world', and provided a convenient mixture of guidance and non-interference that resulted essential under the pressure of the last stage of this thesis.

I am also grateful to the people that set the initial ideas in this thesis: Carles Sierra, Ramon López de Màntaras, Josep Amat, and Francesc Esteva. I offer special thanks to Carles Sierra for following my work and providing suggestions whenever needed. Lluís Godo deserve also special thanks for its mathematical advise.

While at IIIA, I have enjoyed the affectionate support of its people. They have created a comfortable place to work as well as to have discussions, lunch, and 'pica-picas'. I sincerely appreciate the effort of Jaume Agustí for understanding my uneasiness and Pedro Meseguer for increasing my always-low self-confidence. I would also like to thank Dolores Cañamero: she has offered me the most valuable support in both personal and work issues. I cannot omit thanking the Ph.D. students at the IIIA for our conversations. Among all of them I feel specially attached to Jesús Cerquides, Francisco Martín, and Eduard Gimenez: three different worlds that, in spite of complementing my points of view, have always meant a reference. Finally, most recent IIIA visitors, Thomas Dietrich and Peyman Faratin, brought fresh air when needed the most.

During the thesis, I have been participating in several projects. I would like to thank senior researchers that trusted me: Enric Celaya and George

A. Bekey have been remarkably kind and respectful. Carme Torras also deserves mention.

I cannot separate my memories of my stay at USC from many names. First, the person who invited me, George A. Bekey, the professor with an endless number of interesting stories. And secondly, Gaurav Sukhatme, which followed and supervised my work. I also want to thank Maja Mataric for promoting my work among her colleagues and Barry Werger for introducing me to the Pioneer robots. I deeply appreciate the care Kusum Shori took of me, as well as some lab-mates: Jim Montgomery's admirable optimism and friendly support; Stergios Roumeliotis' Mediterranean air, which reminded me of home; and Aswath Mohan's long talks, that introduced me to some aspects of the Indian culture. Of course, I cannot forget my home-mate and friend, Miguel Angel Varón, who introduced me the 'artist' side of Los Angeles.

Finally, I would like to thank my family in my own language:

Tener una familia como vosotros lo significa todo para mi. Vosotros habéis hecho la *maite* que hoy existe, no sólo por haberme dado todas las bases necesarias para ser la persona que soy, sino por haberme apoyado y respetado tan incondicionalmente. Nunca podré agradecer todo lo que se merece el darme un modelo como el que me han dado mis padres, la amplitud de miras de mis hermanas, las preguntas de mi hermano y la vida compartida de Toni. Miro con ilusión hacia el futuro, y a pesar de estar sembrado de dudas, sé que cualquiera que sea el camino que tome, ese camino pasará por vosotros.

Aprendo de mis pasos
sintiendo en mi caminar.

Julieta Venegas (México)

A quienes me han hecho y deshecho.
A todo lo que nunca tuve, y conservo.

Chapter 1

Introduction

This thesis presents some work that resulted from applying Artificial Intelligence techniques into Robotics.

Both Artificial Intelligence and Robotics have been defined in a variety of manners. Here, we cite two definitions that illustrate how close they can be:

> "Artificial Intelligence is the study of computations that make it possible to perceive, reason, and act" (Winston 92)
>
> "Robotics is the intelligent connection of perception to action" (Brady 85)

The interests of both Robotics and Artificial Intelligence concur specially in the research of autonomous robots:

> "Considering a Robot as an active, artificial agent whose environment is the physical world [...] Autonomous Robots are defined as those that make decisions on their own, guided by the feedback they get from their physical sensors" (Russell 95).

In general, robots are assumed to perform tasks in physical environments. Although real environments are very demanding by nature, their difficulties increase significantly when robots become mobile. During the execution of their tasks, autonomous mobile robots have to deal with several environmental aspects:

- Uncertainty about the environment: robot's sensors are imperfect and have a limited range that prevents to obtain global information.
- Uncertainty about robot's actions: on the one hand, there is no guarantee that an action will be accurately executed, and on the other, the effects of an action may change over time.

- The real world is dynamic and complex: it presents a large number of different situations to deal with.

This thesis describes research addressing the map generation problem using groups of autonomous mobile robots. This chapter introduces this problem as well as two other related problems: robot navigation and path planning. The aim of this introduction is to provide a context by giving some definitions and to provide a general view of our work.

1.1 Problem Statement

1.1.1 The Map Generation Problem

In general, a Map can be considered to be a model of a given environment. Nevertheless, this definition might suggest a complete and accurate representation of the environment, and this might imply to specify an impracticable number of features with an infeasible accuracy. Hence, in this thesis, we define a map as an approximate description of some of the relevant features in a given environment. A map does not need to contain all the information of the environment but only those relevant features that can be acquired and recognised. The relevance of the features will depend on the purpose for which the map is needed.

From this map definition it is possible to define the Map Generation Problem through the following question: how to generate a map as accurate as possible considering information that comes from uncertain and limited resources? When these resources are robots, the solution to this problem can be addressed by different approaches depending on the characteristics of the environment, on the robots' sensor and actuators capabilities, and on the subsequent use of the map.

The characteristics of the a priori knowledge about the environment define the map generation process. In this manner, if the environment has been augmented by adding landmarks, mapping means recognising landmarks and establishing relations between them. On the contrary, if the system has no landmarks but an initial approximate map of the environment, the mapping process consists in matching the previous knowledge in the map with actual detections in order to refine the map. And, finally, when the environment is unknown and not augmented with landmarks, there is no direct way of distinguishing the relevant information and, therefore, every piece of information discovered by the robot should be included in the map.

Robot equipment characteristics are also crucial in the mapping process. This is because the nature of the sensors not only constrains the kind of information that can be retrieved but also the imprecision associated with it. For example, robots can know with more or less accuracy their position depending on how they obtain their odometry information (encoders at the wheels for dead reckoning, compass for heading, or GPS for global positioning). At a given position, robots can gather very different kinds of data depending on the sensors they are equipped with. Cameras, sonar sensors, infrared sensors, and laser range finders are different possibilities that require specific methodologies for data extraction and subsequent analysis.

The general settings of each specific mapping problem have a significant influence when choosing the representation of the environmental features and their relations in the map. It is very common to represent indoor environments with *area-based* map representations (Lee 96) such as occupancy grids, certainty grids or probability maps. These representations divide the space into distinct regions with associated properties (which usually describe occupancy). Very often, the size of the regions is constant. This implies a homogeneous space resolution that is not appropriate for extensive outdoors environments with low obstacle density. This reason yields outdoor environments to be represented by *feature-based* maps that pay more attention to obstacles rather than to free-space. They usually represent an environment as a list of primitive features and their properties. Topological maps can be seen as feature based maps including relations among features (arcs in a graph representation).

The representation of the map must also deal with the uncertainty associated with map elements. Although there are several alternative approaches to uncertainty treatment (such as Possibility Theory, Fuzzy Logic, or Evidential Theory), Probability techniques are the most commonly used for both area-based and feature-based maps. Uncertainty is associated with information that comes from robots' sensors. Therefore, since sensors have been classically modelled in probabilistic terms, the probabilistic representation has been widely preferred.

Finally, we also consider that the ulterior use of the map must be also considered when choosing the representation of the map. In this sense, we follow the ecological psychology theory of affordances (Gibson 79), which says that things are perceived in terms of the opportunities they afford an agent to act. In the same manner, we represent maps in those data structures that are more suitable for the posterior use of the map. In general, graph representations are more useful to plan paths than grids. This is because the algorithms for finding optimal paths in graphs are well known, whereas discretized representations use less efficient approaches such as potential fields —which have local minima problems— or reinforcement learning —whose long learning periods cannot be afforded by most autonomous robots applications.

1.1.2 Robots' Navigation

Navigation in this thesis is understood as the policy that an autonomous mobile robot must apply when moving inside an environment.

Navigation becomes a real challenge when the autonomous robot must perform it without having an accurate model of the environment nor an accurate model of its actions and their results. In this manner, robot's actions are not performed exactly as they are expected and their results are not completely predictable nor perceptible. Citing Kortenkamp et al. words:

> "Mobile robots pose a unique challenge to artificial intelligence researches. They are inherently autonomous and they force the researcher to deal with key issues such as uncertainty (in both sensing and action), reliability, and real time response. Mobile robots also require the integration of sensing, acting and planning within a single system. These are all hard problems, but ones that must be solved if we are to have truly autonomous, intelligent systems" (Kortenkamp 98).

In the early stages of mobile robots research, the sense-plan-action paradigm was the only approach to robot navigation. The link between mobile robotics and Artificial Intelligence was probably first forged with SHAKEY, a robot developed at the Stanford Research Institute in the late 1960s (Nilsson 69). By the 1980s Artificial Intelligence research was beginning to produce large software systems that, from a starting position, could take a goal location and generate a sequence of actions that reached the given goal. This required a model of the world, which was complex and often had to incorporate large amounts of domain knowledge. Moreover, each step of the generated plan was passed to the control level of the robot for execution, which meant that the plan had to include actions down to the actuator level. As a consequence, the amount of data moving from the sensors to the centralised computing resources was significant, and the computational overhead did not allow to synchronise the world representation with the real environment.

In 1986, Brooks (Brooks 86) showed that behaviours similar to those observed in simple animals —such as insects— could be produced with very little, if any, internal state. The subsumtion architecture was designed to make it easier to build these robots so that the most appropriate behaviour would be used at each moment. These behaviours would implement tight sense-act loops using asynchronous finite-state machines. Higher level behaviours would dominate (subsume) low-level behaviours by suppressing their output to the actuators. The idea was to build a robot that accomplishes tasks through behaviours that ran in parallel and are arbitrated by simple strategies based on priorities. The world model was thus distributed among the behaviours, with only the relevant part of the model being processed for each behaviour.

Nowadays, most applications lie somewhere between behaviour-based approaches and world models, increasing the emphasis on sensing and acting and reducing the emphasis on planning. Arbib and Arkin (Arkin 87) pioneered the new paradigm from a cognitive science perspective within the robotics community, calling it action-oriented perception.

1.1.3 Definition of the Path Planning Problem

In the context of an environment, if an autonomous robot —initially located at a certain position— is assigned the task of reaching a goal position, then it needs to describe a certain trajectory in order to fulfil its task. Considering this trajectory as a Path, and considering as well the fact that this path must be somehow planned, we encounter the Path Planning problem. It is a well-known problem for which, given an environment and two positions belonging to it, it is required to specify the trajectory connecting these two locations. The entire trajectory must belong to the free space area of the environment, and usually, it must fulfil some additional requirement such as to be the shortest path or to be the safest one. When the energetic resources of the robots (i.e., batteries) are limited, short paths are essential. On the other hand, safe paths have been habitually planned for manipulator's applications, which try to maximise the distance to the obstacles in order to avoid collisions.

The path planning problem has specific solutions when the map (in the sense that subsection 1.1.1 defines) of the environment is known (the path planning section —8.1— introduces several alternative approaches). Nevertheless, how to plan over a partially known environment remains as an open issue.

1.1.4 Sources of Uncertainty

Different kinds of sensors obtain information about the environment or the robot's state by means of different processes. This variation in the perceptual acquisition of information means that noise alters the measurements under different circumstances. Therefore, the resulting uncertainty must be considered specifically for each kind of sensor.

When evaluating the information that a robot can obtain from its environment, there are two main aspects to take into account:

- Unreliability: to what extent what the robot is sensing is real, and
- Imprecision: how accurate is the obtained information (this depends both on the robot's positioning accuracy and the precision of its sensors).

In this subsection we briefly describe the most commonly used sensors and comment the kind of associated uncertainty they have.

Ultrasonic Range Finders

Ultrasonic Range Finders are commonly known as sonars (Sound Navigation and Ranging). Generally, an ultrasonic sensor is composed by two fundamental elements: an acoustic transducer, which generates a packet of ultrasonic waves, and a ranging circuit board whose task is to detect the resulting echo of the signal. The time delay between the transmission and reception allows to compute the distance of the sensed obstacle.

This measuring process is affected by three basic sources of uncertainty (Poloni 95):

- The radiation cone does not have a zero width (in fact, it is usually about 25 or 30 degrees wide). Inside the cone, it is not possible to determine the angular position of the object that originated the echo (see examples a), b), and c) in Figure 1.1).
- Radial resolution is finite: in general, the accuracy remains high for a certain range that usually goes from 10 cm to 10 metres.
- Specularity: if the incidence angle is larger than a critical value, the sensor reading is not significant because the beam may have reached the receiver after multiple reflections, or as part d) in Figure 1.1 shows, it could even be lost.

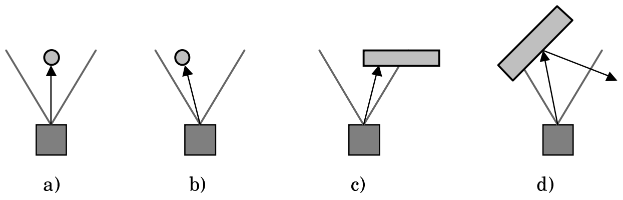


Figure 1.1: Typical uncertainties with ultrasonic sensors. The objects in a), b), and c) give the same measurements. The signal in d) is lost.

Infrared Sensors (IR)

Similarly to sonars, infrared sensors are radiant transducers (transducer is a sensor plus associated circuitry) with the same transmission-detector system that evaluates the distance to detected obstacles. The main differences are that they emit light instead of sound and that they cover a smaller range. A LED (Light Emitting Diode) emits the infrared signal, which is affected by the colour of the object it encounters: if the surface of this object is black most of the emitted light will not be reflected so that the object will not be detected. In this manner, since white surfaces are the ones that reflect light more effectively, robots with infrared sensors are more suitable for environments mainly occupied by white (or light coloured) objects and walls.

Laser Scanners

Laser scanners are active sensors that emit a low-powered laser beam that is scanned over a surface. Through techniques such as phase-amplitude modulation, the distance to the individual points can be computed with the net result of an array of image points, each of which has an associated depth. In effect, a three-dimensional image is obtained. Despite their very high cost, many current implementations of these devices present mechanical instabilities and sparse data sampling at long distances.

A typical configuration of a laser is a lower-cost linear array laser scanner known as *Structured light vision system*, (Fabrizi 99) which is composed by a laser light emitter and a video camera. The emitter generates a laser beam that goes through a cylindrical lens to create a plane of light. Such plane is emitted in a circular sector and produces a bright line (object line) when it hits an object. The camera is placed so that its horizontal scan lines cross the object line. Exploiting the geometrical transformation between the camera and the plane of light co-ordinate frames, it is possible to compute the distance associated with each pixel. The performance of the structured light sensor may be affected by various phenomena such as intense ambient light, the colour of the reflecting surface, camera occlusions, or absence of the object line (due to a mirroring of the laser beam).

Cameras

Together with sonars, cameras are the most commonly used sensors in autonomous mobile robots. Historically, cameras have been used to recognise previously known features that are used as landmarks (Lazanas 95). They have also been widely used in autonomous vehicles, in order to track the lines or edges of roadways (Kiy 95). Image processing is a research area by itself that goes beyond the scope of this thesis, so we are not going to get into the details. Just comment that in order to perform certain tasks (such as navigation, manipulation, or recognition) it is necessary to extract 3-D information from the environment. Usually, this is implies:

- Segmentation of the scene into distinct objects, and
- Determining the features (position, orientation, and/or shape) of each object.

The objective of image segmentation is to separate the components of an image into subsets that correspond to the physical objects in the scene. The image segmentation methods assume that the objects have smooth homogeneous surfaces that correspond to regions of constant or smoothly varying intensity in the image and that the intensity changes abruptly at the boundaries (Nevatia 86).

The appearance of an object in an image can be modelled using appropriate two-dimensional shape representations, so that instances of such objects can be recognised and quantitatively inspected by comparing their actual appearance in an specific image with their expected appearance as determined by the model (Davis 86).

Extracted information can be highly affected by shadows or occlusions, and, although stereo vision (two calibrated cameras) allows the generation of depth maps (which specify distances to objects and increase accuracy), the extraction process becomes expensive and computationally demanding. Therefore, cameras are usually combined with other sensors. Otherwise, they tend to focus on recognising specific views as the purely-vision based approach in (Franz 97), which obtains 360° horizontal views by using a conical mirror mounted above a camera pointing to the centre of the cone.

Global Positioning Systems (GPS)

Information positioning constitutes a key information for mobile robots. Global Positioning Systems (GPS) use three or more satellite signals to compute the position of the robot. This position is given in world's co-ordinates in terms of latitude, altitude and longitude and it is computed based on the travelling time of the GPS signals and triangulation. Differential GPS use an additional ground-based transmitter that allows to obtain a positional resolution of submeter ranges. The position information has the guarantee that the real robot location is included within a fixed area around the given co-ordinates, so that the error is not accumulative. Apart from their cost, their main problem is that the satellites' signals can be lost. This makes GPS more suitable for outdoor applications and forces the robots to have some kind of odometry capability to be used in case the signals are not received.

Wheel Encoders

Various kinds of encoders are available for positioning purposes, most of them count the number of wheel rotations, whilst others recognise the direction of rotation. The former ones work on a photoelectric principle. A LED emits light through an indexed code wheel (which is rigidly mounted on the motor shaft). And the receiver (phototransistor) converts the light/dark pulses into electrical signals, which are subsequently treated in order to compute the number of wheel rotations.
In this manner, it is possible to compute the position of the robot by keeping track of all its movements (both turns and displacements). This is usually known as dead reckoning, which was originally derived from *ded*uced *reckoning*. The main encoders' problem is that, although their resolution can be very high for manipulator's applications, they accumulate a significant error in mobile robots, mainly due to drift and wheel slippage.

1.2 The approach of this Thesis: Multi-robot Mapping Solutions

1.2.1 Motivation

To generate maps by means of mobile robots is an open problem that has been addressed from different points of view. In this section we are not going to describe what was done before 1994 (when we started the work of this thesis) because the related work section in the last chapter is devoted to that. Nevertheless, we would like to emphasise the conclusions we extracted after studying the ongoing research:

- On the one hand, all the previous map generation approaches involved the exploration of the environment by means of a single robot. As a consequence, the flexibility and accuracy of the whole system relied on the fault tolerance and the precision of one robot.
- And, on the other hand, each approach focused in some specific aspects of the problem, (such as the map representation or the information acquisition), but there was a lack of a global study of the problem, and therefore, some of the important issues in the solution were simply taken as assumptions.

These two aspects represented two great challenges for us:

- First, the distribution of the map acquisition phase.
- And second, the study of the problem from a rather general point of view, taking into account all the relevant aspects for the definition of the settings (or bases) of each specific mapping application.

The distribution of the map acquisition phase has been faced through the use of several robots.

This allows to increase the robustness of the map generation process. This is because the quantity of information that the map contains degrades gradually with the number of robots that do not succeed in their exploration task. Moreover, since the error associated with the gathered information accumulates along the exploration, it is better to have k robots that explore an area during a certain period of time than to have a single robot that explores k times this period.

Multi-robot approaches can be characterised by the use of homogeneous or heterogeneous robots. Heterogeneity is usually related to specialisation, and can refer both to the use of different robots or to equal robots performing distinct tasks. In this thesis we present two basically homogeneous approaches and a third one that is heterogeneous.

The study of the problem has been done trough a compilation of several new and alternative approaches to different instances of the problem.

The specification and implementation of the three solutions we propose has provided us with the experience necessary to specify the problem as it is presented in the previous section. We claim that, when facing the mapping problem, it is essential to make an explicit characterisation of its starting assumptions and problem settings (type of environment, number of robots, gathered information, etc.). This is particularly important when one has to deal with real problems, i.e., problem settings provided by the context of the projects, as in our case, as opposed to problem settings designed to test a given approach. In order to avoid considering unrealistic assumptions, we have tried to make explicit the concrete features that are specific to each problem settings and that therefore had a strong influence in the choice of each particular approach we present.

The analysis and specification of the problem settings is a first step towards a better understanding of the relation between the nature of the problem and the approach taken, so as to better assess the suitability of the approach and the significance of the results obtained.

This sometimes appears to be hidden when analysing the results of a specific mapping approach. Obviously, in order to obtain concrete solutions, the analysis cannot be done from a strictly general point of view. Rather on the contrary, our aim in this thesis is to use specific cases to illustrate that some solutions are more suitable for specific problem settings than others.

We apply different Artificial Intelligence techniques for each of our approaches.

The given solutions do not present new Artificial Intelligence techniques but expand the potentiality of the existing ones by providing original applications in the Robotics field.

1.2.2 Different Approaches to the Multi-Robo t Mapping Problem

This thesis presents three new and alternative approaches to solve the map generation problem when using several autonomous robots. These approaches have been specified on the basis of a number of aspects we consider essential for the definition of the mapping problem settings. We present them as a list of questions that should be asked when facing any specific mapping problem for the first time.

1.2.2.1 We Need to Ask the Right Questions

We have already commented that when facing this problem there are several aspects we need to define:

- How will the environment be considered? (Indoors or outdoors? previously augmented with landmarks, partially known, or completely unknown?).
- What kind of robots will be used? How many? Will the group be homogeneous or heterogeneous?
- How will the robots be equipped? How will the robots behave when gathering information (i.e., exploring)? What will be the co-operation level?
- What kind of information will the robots gather? What will be the main source of uncertainty associated to the obtained information?
- How will the map be represented? (Feature-based or area-based? Will the representation be able of including uncertainty and being simultaneously suitable for the task?
- How are we going to treat the resulting map? What is the main task of the robot group?

1.2.2.2 Three Different Answers

This thesis attempts to answer these questions through the presentation of three approaches together with the motivations that yield us to apply different solutions depending on the characteristics of each specific problem.

Representation of Orthogonal Indoor Environments by means of Fuzzy Techniques

In the first approach we have a group of robots (developed at the UPC[1]) that explore an unknown environment. This environment is considered to be indoors and orthogonal. The robots explore by moving randomly in free-space and following walls —or obstacle edges— when detected. The information that will be used to generate the map will consist of the portions of walls or obstacles that have been followed by the robots. In order to filter spurious sensor readings that come from non-existing features, we only consider those features that, after being detected, have been indeed followed. Our robots are small, compute their position by dead-reckoning, and use infrared sensors to follow walls from a close distance. This means that the uncertainty related to the position of a followed wall is mainly due to the error associated with the robot position. The robots co-operate by sharing the information about the followed features when two of them meet. Finally, at the end of the exploration, they communicate their information to a host computer, which is in charge of generating a map of the environment.

Considering these settings, the map representation that has been chosen is the feature-based one. More concretely, we represent followed features as orthogonal segments (i.e., segments that can only be vertical or horizontal). These segments are imprecise, and we represent their imprecision by means of fuzzy sets (which are computed on the basis of the accumulated robot position errors). Although we have previously commented that indoor environments are generally modelled by means of area-based maps because indoors have a limited size and a significant density of features, we use feature-based because the number of followed features is reasonably small and because this is a kind of 'symbolic approach' that allows the host to reason about the gathered information. Hence, the imprecise segments can be fused, combined or modified in order to generate higher level features such as corners or doors.

The representation of the uncertainty by means of fuzzy sets provides a simple fusion operation and, unlike the Probabilistic approaches, does not require a large number of well distributed data to be available (which is usually not available during autonomous robots' navigation).

[1] UPC: Technical University of Catalonia, ESAII: Department of Automatic Control.

Using Possibility Grids for Structured Indoor Environments

The second approach is based on the simulation of the robots in the previous approach. In this case, we have developed a simulator system which includes a robots' behaviour-based navigation control. The settings are very similar, there are only two main differences. First, the orthogonal assumption about the environment has been relaxed in order to include oblique features. And second, the task is not only to obtain a map but to plan paths towards less explored areas. The simulation system also includes the behaviour-based navigation control required for the robots in order to follow the planned paths and apply reactivity when necessary.

In order to plan paths, the information of free-space is as relevant as the information about detected and followed features. To include trajectory information implies that a much higher density of information should be included in the map representation. Therefore, the main concern is now not to reason about the features of the map but to distinguish between occupied, free and unknown areas in the explored environment. This, together with the fact that we are still indoors, makes more suitable the occupancy grid approach to represent the map. In our case, we have chosen Possibility Theory to represent robot trajectories and detected features. Robot trajectories are modelled through possibility distributions and detected features (that have been followed) are modelled using necessity value distributions. Again, the main source of imprecision is the robot odometry error, which is used in the generation of these distributions. Possibility theory is specially useful because it assigns, for each cell in the grid, two dual values (necessity and possibility degree of being occupied). Contrarily to Probability, which cannot model ignorance for the unexplored areas, these values allow to represent ignorance in addition to occupancy and free-space.

Robots store their gathered information in terms of trajectory and 'wall' segments. These segments are discretized into certainty values that are assigned to adjacent cells in the grid. This segment discretization simplifies the certainty value combination because it becomes local to each cell. And, although there is not a explicit segment representation, we are still able to find the boundaries of wall segments and extend them in order to increase the coverage of the environment.

Finally, the information in the grid is transformed into a visibility graph which allows to obtain the shortest (with some safe restrictions) paths to goal positions. The path is computed with the A* algorithm, an optimal AI search technique based on heuristics. We also study how extended maps yield to safer paths that decrease the use of reactivity when following them.

Using Symbolic Grouping for Outdoor Environments

The third approach involves outdoor environments and was developed during my stay at USC[2] joining the Taskable Heterogeneous Robot Colonies project. In this project, the colony is formed by several ground sonar-equipped robots and an autonomous flying vehicle that has a camera pointing to the ground. Robots obtain and broadcast information about obstacles in the environment so that each of them can generate the same map (or similar, in case of communication problems). Ground robots' main task is to plan paths towards a goal position. These plans must be updated each time a new obstacle obstructing the path is detected.

Considering that we have outdoor environments (that is, large environments with low density) and that obstacles are represented as polygons, the feature-based maps are naturally chosen. We build maps by grouping obstacle polygons into higher level structures with simple shape: the obstacle areas. Obstacle areas simplify and group information. Therefore they allow the generation of tractable visibility graphs, and thus a simple graph update and path planning (simpler than if the graph was generated directly from the obstacle polygons). The generation of the map is incremental: new polygons are included into the map obstacle areas, forcing an update of the visibility graph. Nevertheless, the polygon information is not lost, so that we can plan paths at different levels of detail (obstacle area level or polygon level).

When considering the uncertainty associated with the obtained information, the main concern becomes to discern among the received polygons: which ones correspond to real obstacles from those others that are actually the result of shadows or other colour changes in the images. We represent this uncertainty by means of certainty degrees associated with each polygon. These values are aggregated in the obstacle areas and are proportional to the polygon areas and to the sensor reliability.

1.2.2.3 Assumptions

One of the aims of this thesis is to present map generation approaches that consider and analyse all the aspects in the problem, without hiding basic assumptions that could affect the feasibility of the approach. Nevertheless, some form of assumptions are almost unavoidable. In this thesis, we assume two aspects that are common to the three approaches:

[2] USC: University of Southern California, Robotics Lab.

- The environment is considered to be completely unknown and not augmented with landmarks, but easily passable and mainly reachable for robots equipped with wheels.
- The environments can be properly modelled by 2-dimensional maps. In two approaches, maps represent objects at the plane defined by the sensors, and in the third approach, aerial views give planar projections of all the objects in the environment.

1.2.3 An Schematic View of Our Approaches

At the beginning of this section we have emphasised the importance of asking the right questions when defining a specific solution for the mapping problem. The following Table 1.1 summarises the answers provided by each of our three approaches.

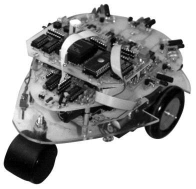
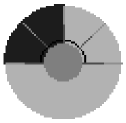
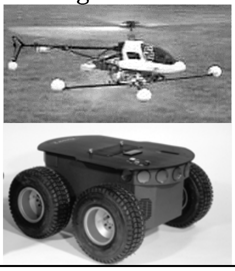
	Approach 1	Approach 2	Approach 3
Environment	Unknown Indoor Orthogonal	Unknown Indoor Structured	Unknown Outdoor
Group of robots	Homogeneous 	Homogeneous Simulated 	Heterogeneous 
Tasks	Exploration Map generation	Exploration Map generation Path planning Path Following	Map generation Path planning
Co-operation	Share information Host communication	Share Information Host communication	Broadcast Information
Information	Followed walls	Robot trajectories Followed walls	Obstacle polygons
Uncertainty	About location	About location	About existence
Uncertainty representation	Fuzzy Sets	Possibility/Necessity values	Certainty degrees
Map approach	Feature-based	Area-based	Feature-based
Map elements	Imprecise Segments Corners Doorways	Grid cells Visibility graph	Obstacle polygons Obstacle areas Visibility graph
Map generation	Segment fusion	Value combination	Polygon grouping
Map treatment	Map completion	Map extension Path planning	Path planning Path update

Table 1.1: Brief description of our three approaches: compact answers to the mapping questions.

1.3 Distribution of the Parts and Chapters

Each one of the three approaches is respectively presented in the first, second, and third parts. The last part concentrates on the conclusions of the thesis.

Part I

Chapter 2. This chapter introduces our first approach to map generation. Initially, it specifies the robot characteristics. The second section describes how the robots explore the environment, what kind of information they gather and how do they co-operate. Finally, the third section corresponds to a statistical error analysis that will be used to represent the uncertainty in both, the first and second approaches.

Chapter 3. The map generation process is the focus of the third chapter, which constitutes the central part of the first approach. Initially, it introduces the basic Fuzzy Logic notions in order to give the background to the imprecise segment definition, which is subsequently presented. These imprecise segments are the basic elements in the map representation, and there are two processes applied to them: fusion and completion. The chapter ends with the algorithms implemented for both processes.

Chapter 4. The ultimate chapter of the first part presents the results obtained by the indoor feature-based mapping approach. It details the used environments, different navigation strategies, the gathered information and the map generation (including fusion and completion). Finally, it includes some further remarks related to the random exploration strategy, the development of the approach and some final conclusions.

Part II

Chapter 5. The second approach is completely based on simulation. We have developed a simulator application that allows us to define and explore different environments. Several robots can simultaneously explore the environment where they have been included. This fifth chapter is a rather technical description of the simulator, it includes the description of specific aspects of the simulated robots such as how they sense, how they execute actions, their odometry errors, the information that they gather, how to interpret their trajectory lines on the screen, etc.

Chapter 6. Still in the simulator, the robots navigate based on the co-ordination of several basic behaviours. This chapter specifies the control architecture used for robot exploration and describes the rules used to implement each basic behaviour as well as the ones that specify the behaviour co-ordination. It ends with the presentation of the exploration results.

Chapter 7. Back to the map generation issue, the penultimate chapter of this part describes the basic features of the second mapping approach. It introduces the Possibility Theory and how it is used to represent and combine both the occupancy and the free-space information in the grid map. It also compares the possibilistic model with the probabilistic and evidential models. And finally, it presents the map generation results.

Chapter 8. The ultimate chapter of this part contains the specification of the map refinement process. It first introduces the wall extension process and the corresponding results. Second, it describes the graph extraction, the path planning processes, and their results. As a separate section, the robots' path following strategy is presented. As for exploration, this strategy is based on similar basic behaviours that allow to follow a path and react when unexpected obstacles are encountered. Finally, there is a discussion about how extended maps yield, in general, to safer paths that require less use of reactivity when being followed.

Part III

Chapter 9. The description and results of the third approach are condensed in this ninth chapter. It includes the specification of the input information, how it is grouped in order to generate the map and how is it used to plan and update paths towards goal positions. The results are distributed along the sections. The chapter ends with the description and treatment of the uncertainty associated with the obtained information.

Part IV

Chapter 10. This chapter constitutes the last chapter of this thesis. Hence, it is dedicated to provide the conclusions and contributions of this thesis. It presents as well the related work and a comparison with our work. Finally, it ends with a brief description of what could be the future work.

Part I: Representation of Orthogonal Indoor Environments by means of Fuzzy Techniques

Chapter 2

Troop of Small Autonomous Robots

The first part of this thesis presents a part of the Autonomous Navigation Troop (A.N.T.) project. The main objective of the A.N.T. project (Amat 95) was to model an unknown office-like environment. More precisely, this work presents a solution to the map generation problem for cases in which the environment is structured and indoors.
The solution is given generally in terms of a Master-Multi-Slave approach, where a troop of small autonomous robots could be considered as being 'slaves' of a host computer (the 'master'). This approach consists of two main steps:

1. A co-operative and distributed exploration of an unknown environment using autonomous robots.
2. Incremental generation of environmental map by a host computer which uses fuzzy techniques to combine the information received from the robots.

This chapter concentrates on the first in the approach above. First section gives more details about the troop of robots and the task. Section 2 focus on the hardware characteristics of the robots: their architecture and their sensors. The third section explains how the robots explore, the information they gather, and how they communicate. Finally, the last section studies the robots' odometry errors and their accumulation. The measurements of the errors, their statistical analysis, and how they are modelled are also introduced in the last section. The next chapter will show how this analysis will be used to model the imprecise location of obstacles and walls by means of fuzzy techniques.

2.1 Specification of the Team and its Task

In the ANT project a troop of low cost, small autonomous robots has been developed for exploring an environment that is unknown but easily passable. These robots follow the already classical line of insect robots (Alami 93, Brooks 86, Brooks 91).

The goal of these autonomous robots is to obtain partial information about a vertically and horizontally arranged (orthogonal) environments during their exploration runs. This information is subsequently supplied to a host computer which computes the corresponding global map. The environment is completely unknown for the robots. Nevertheless, there are some assumptions about the environment that are taken —although they do not concern the robots. On one hand, we assume that most parts of the environment are easily passable if we want the robots to succeed in their task of exploration. On the other hand, although the environment is also unknown for the host computer, it considers some assumptions that simplify the process of building the map: it is indoors and orthogonally structured.

Using this multi-robot strategy and a host computer to generate a model of the environment, a better efficiency is expected than that which would be obtained based only on a single expensive robot. The Master-Multi-Slave strategy is highly distributed, thereby providing two main advantages: price and robustness. We propose a distributed approach based on various small cheap robots rather than a centralised approach that relies on the exploration task using a unique expensive robot. To guarantee the success of only one explorer robot is especially difficult for an unknown environment. In our distributed approach, the autonomous robots co-operate by transferring the perceived environment to each other when they meet. In this manner, not all the information of the non-returning robots is lost provided that they had encountered other robots that were able to return.

The behaviour of these small autonomous robots could be seen as a metaphor of the behaviour of ants. We did not try to follow any biological model, but in some sense we were inspired by two aspects of ants: first, how they spread when looking for sources of food, and second, how they communicate when building a path from their nest to the food. Similarly, our robots explore randomly and transfer information to each other when they meet. Nevertheless, robot navigation strategy is designed so that robots show three different random behaviours —calm, normal, and anxious. These different behaviours are used in order to increase the coverage of the environment.

2.2 Hardware Characteristics

Each robot has been designed with the aim of being small and cheap. They must have a high level of autonomy and be endowed with a low cost processor to memorise a map of the perceived environment. All these requirements have lead to a solution consisting of small robots with three wheels. Two of them are steering wheels, having independent motors, and the third wheel is passive.

The robots environment perception system and the communication with the host, or with other robots, is based on infrared (IR) impulse modulated sensors. Since some of the robots may not be able to return to deliver their map to the host, the following communication process is established: when two of them meet while at an exploration run, they exchange all the information they have acquired from the environment so far. In this way, when a robot reaches the host, it delivers both, the information acquired during its own run as well as the information obtained from other robots that it has encountered. This communication process permits the collection and transfer of all the information of those non-returning mini robots, by those that return to the host.

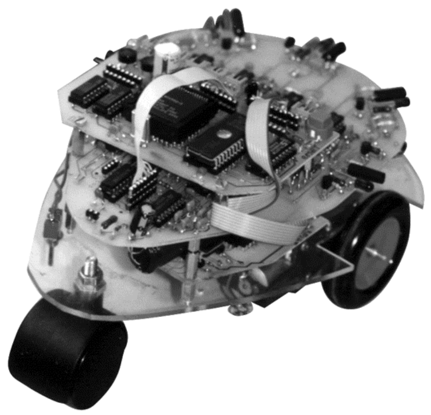


Figure 2.1: Autonomous Mini-Robot.

Robots are composed of the following hardware specification. Each robot is 21 cm. long and 15 cm. wide (see Figure 2.1). Two 5 cm. driving wheels function to permit to pass over some small obstacles such as carpets or electrical wires. The robots can reach a maximum speed up of 0.6 m/s., and since the battery has about one hour of charge, each robot could, in principle, explore a maximum distance of about 2000 m.

2.2.1 Architecture: Functional Modules

The design of control unit in each robot has been constrained by both the simple hardware described above, and the need for a behaviour which is sufficiently smart in order to navigate efficiently. Furthermore, the robot had to be based on a flexible hardware to allow for experimentation of navigation and control strategies. These requirements have resulted in a design which contains three different functional modules (see Figure 2.2):

- A navigation module, that generates the trajectory to be followed;
- A steering module, which controls the motors in order to follow the generated trajectory; and
- A perception module, that acquires information of the environment by means of IR sensors. However it is possible to replace this module by other modules adapted to different types of sensors.

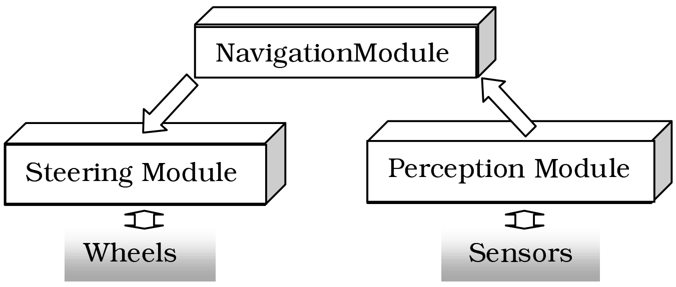


Figure 2.2: Three functional modules in our mini-robots.

The computer used to implement the navigation control unit is a 80C186 with 1 MB of RAM used to store the environment map perceived by the robot and, in case it has met other robots, the respective communicated maps.

The steering control module is implemented on a 80C552 and operates with high resolution since each encoder corresponds to a displacement of only 2 mm. A special control is needed to follow walls and obstacles, and this is done by micro-corrections of the robot orientation in order to keep the robot parallel to the followed wall. This kind of controller is known as the bang-bang controller and results in a zigzag trajectory.

2.2.2 Sensing Capability

Each robot is equipped with the following sensors:

- Impulse generators at each wheel for odometry, which continuously provide the information about the number —and fractions— of turns that each wheel performs. In this manner, when both steering wheels move in the same direction, these encoders give the distance that the robot is covering. Analogously, when the wheels move in opposite directions, the turning angle of the robot is obtained.
- Five infrared (see page 7) proximity sensors for frontal obstacles detection and for wall following. These infrared sensors are distributed in the front side of the robot, considering the front as the origin, they are at 0, ±45º and ± 90º. These sensors provide two possible readings: near and far, which correspond to 10 cm and 20 cm respectively.
- Two proximity sensors oriented to the ground for the detection of the terrain horizontal discontinuities.
- Safety micro switches for protection against collisions.
- One omnidirectional IR Emitter/Receiver sensor to detect the presence of other robots and to transmit data —with a modulated FSK signal that transmits data at 9.600 bits/s.
- One IR Emitter/Receiver with a scope of 90 degrees —going from the frontal axis of the robot to its left side—to generate both a priority signal (right hand preference) and to situate other robots in the neighbourhood.

2.3 Robot Navigation

The navigation system incorporated to each robot has a partially random behaviour. Although our environment has right angles and therefore, ±90º turns would be enough to move into it, the robot also make ±45º turns to increase its random movement choices. Robot movements consist of straight displacements (rectilinear movements) followed by turns on themselves (i.e., spins that change their direction without changing their position) that allow the robots to redirect themselves.

Turns are determined by the value of a turning probability. Different probability values ($P_1 >> P_2 >> P_3$) correspond to three qualitatively differentiated behaviours:

- $P_1 \Rightarrow$ robot with "anxious" behaviour.
- $P_2 \Rightarrow$ robot with "normal" behaviour.
- $P_3 \Rightarrow$ robot with "calm" behaviour.

When a robot finds an obstacle, it turns in order to place itself parallel to it and to follow it during a random distance —which depends on its behaviour. The turn can be done to the right or to the left based also on a fourth probability value P_4. Robots having a probability $P_4 < 0.5$ will show a tendency to turn to the right more often than to the left, whilst the robots having a probability $P_4 > 0.5$ will behave inversely. Therefore, the robots of the exploration troop do not show identical behaviours. They behave in six qualitatively different ways corresponding to the different combinations of turning probabilities with turning tendencies.

During the exploration run, robots store their trajectory and make an estimation of their localisation error. When, for a given robot, this error is bigger than a previously fixed threshold, this robot returns towards its starting point. In order to ensure its returning to be as safe as possible, the robot follows its own trajectory in inverted order and eliminating loops. That is, the robot deletes from its trajectory all sub-trajectories starting and ending at the same point.

2.3.1 Partial Map of each Robot

A partial map is the representation of the information that a robot gathers during its exploration. This information, or partial map, is stored and incrementally generated by the robot, and consists of a representation of its trajectory together with detection information. Since robots movement can be seen as a sequence of rectilinear movements and turns, trajectories are represented by a sequence of robot turning positions. Turning co-ordinates define consecutive extremes of trajectory segments.

Therefore, we can redefine a partial map as being a sequence of robot trajectory segments with associated information about which ones correspond to wall following trajectories. In addition, if it is the case that a robot follows a wall and detects the end of the wall (it may be due to a corner or a door frame), this ending position is stored as singular point. Both pieces of information (i.e., detection and singular points) are stored as labels of the trajectory segments. Detection labels specify the side of the detection, that is, if the robot was following the wall on its right or on its left. This label allows the host to compute the corresponding detected wall segment —it has the same length as the trajectory segment and their co-ordinates are parallel. Furthermore, a trajectory segment having labels of detection and singular point means that the last segment co-ordinate corresponds to the singular point. Notice that there is only one possible singular point associated to one wall segment in the partial map. (Next chapter will show how the wall segments can receive further treatment and result in a segment with any number of singular points).

2.3.2 Communication Between Robots

A robot uses its omnidirectional IR Emitter/Receiver sensor to detect another robot in its vicinity. When this occurs, the robot changes its presence signal into an attention signal and stops. If the other robot detects this signal it will also stop allowing the hand-checking (protocol communication establishment) to take place. Both robots communicate with each other the partial map by a full duplex transmission. After the transmission, the robots orient themselves to follow their previous trajectories. During this reorientation, if a robot detects its partner with its 90º scope sensor, it avoids a possible collision by turning 90 degrees to its left (Figure 2.3 depicts this situation).

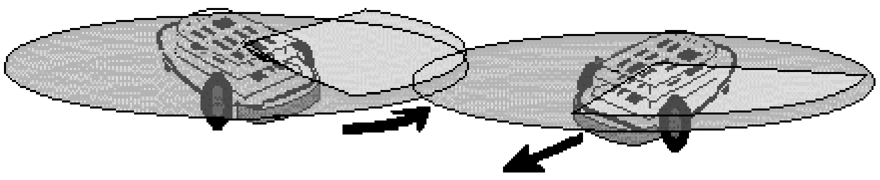


Figure 2.3: Two robots ending their communication process. The left one must turn left to avoid the other.

If a robot meets two other robots that are already communicating each other, this robot detects their communication signal —which is different from both the presence signal and the attention signal. In such a case, it inhibits its presence signal in order to do not interfere the already established communication and it turns until the 90º scope sensor stops detecting any signal.

2.4 Error Analysis

Section 1.1 introduced the hardware specification of the robots. There, it is explained that the position of a robot is obtained from the movement of the wheels. This kind of computation is called dead reckoning. Error appears in the dead reckoned position mainly because of wheel slippage (Kortenkamp 98): there is a difference between the movement of the wheels —the steering— and the distance that the robot is actually covering. The main characteristic of this odometry error is that it is accumulative. And this is mainly what this section will analyse. This is a key aspect because all the information that the host computer uses to build the map of the environment comes from the robots, and all the information they gather is associated to the place the robots ‘think’ they are —i.e., their computed location. Of course, the information is also subject to the error of the sensors, and therefore, they are also discussed at the end of this section.

2.4.1 Sample Description

With the goal of studying the position error of each robot due to the imprecise odometry and to the imprecise steering, we have performed an analysis based on experimental data obtained from real robots. More specifically, we have performed a number of trials of four different displacements:

1. Covering a distance of 3 meters running straight;
2. Covering a distance of 6 meters running straight;
3. Making a 45-degree right turn —which is achieved by moving the wheels in opposite directions the same number of turns that would be needed to cover a distance of 10 cm if going ahead— and afterwards, following a 2.9-meter straight trajectory;
4. Making a 45-degree left turn and follow a 2.9-meter straight run.

Table 2.1 shows the results of all the performed trials for each displacement. The first column contains the number of trial, and the rest of columns represent the positions where the tested robot ended after performing each kind of displacement. The co-ordinates of the given positions are presented in centimetres and take the origin to be the theoretical ending position —that is, the position were a robot without any odometry error would end. In other words, the numbers correspond respectively to the robot displacements in the x and y axis. Taking the first cell of the table as an example, the information that we should conclude is that, after running 3 m, the robot did not ended in the expected position, but 11.1 cm on its right and 3.5 cm too far.

# trial	3 m ahead	6 m ahead	45º right turn & 2.9 m ahead	45º left turn & 2.9m ahead
1	(11.1 , 3.5)	(-22.7 , 1.7)	(-18.8 , -8.4)	(-18.5 , -6.4)
2	(8.6 , 2.3)	(-15.2 , 3.7)	(-17.3 , -2.9)	(-15.3 , -2.5)
3	(10.0 , 1.4)	(-12.0 , 1.7)	(-15.8 , -5.1)	(-13.4 , -4.9)
4	(7.8 , 1.3)	(-9.2 , 4.9)	(-14.1 , -1.2)	(-13.8 , -1.5)
5	(4.2 , 1.0)	(-3.6 , 2.6)	(-13.4 , -2.5)	(-13.2 , -2.8)
6	(3.1 , 0.9)	(-3.5 , -2.1)	(-12.5 , -3.7)	(-12.6 , -1.0)

# trial	3 m ahead	6 m ahead	45º right turn & 2.9 m ahead	45º left turn & 2.9m ahead
7	(0.6 , 2.1)	(-0.4 , 0.7)	(-11.6 , -0.2)	(-12.2 , 0.4)
8	(8.9 , -0.7)	(0.5 , -2.0)	(-10.3 , -1.3)	(-11.6 , -1.7)
9	(6.8 , -0.9)	(4.7 , 2.0)	(-9.0 , -2.6)	(-10.2 , -3.3)
10	(3.6 , -1.1)	(4.3 , -2.4)	(-9.3 ,1.2)	(-9.6 , 0.4)
11	(-3.7 , 2.7)	(4.7 , -1.1)	(-8.3 , 0.7)	(-8.9 , -2.2)
12	(-1.7 , 1.95)	(6.7 , -1)	(-6.5 , -1.3)	(-6.5 , -1.9)
13	(-3.3 , 0.3)	(6.2 , 1.3)	(-5.0 , -1.5)	(-7.7 , 2.8)
14	(-5.6 , 1.1)	(7.0 , 2.0)	(-5.1 , 2.4)	(-2.0 , 3.5)
15	(-12.3 , 1.1)	(15.9 , -1.4)	(-3.1 , 0.9)	(0.1 , -1.2)
16	(-2.5 , -0.4)	(21.3 , 2.8)	(0.1 , -1.6)	
17	(-1.2 , -1.3)	(25.2 , 5.8)	(0.1 , 2.3)	
18	(-11.6 ,-1.5)		(-11.4 , -3.7)	
19	(8.2 , 2.6)			
20	(-10.2 ,-1.8)			
21	(-6.1 ,-2.9)			

Table 2.1: Performance of a number of trials for four different robot displacements.

2.4.2 Statistical Analysis

With the data obtained, we have performed an statistical analysis (Savage 72). The most important aspect to study in this sample is if the values from the x co-ordinate and the values from the y co-ordinate can be respectively associated to two independent random variables: X and Y. A negative answer to this question would mean that the displacement in one direction is related to the displacement in the other direction (i.e., the sample corresponds to correlated values of a bi-variant random variable). This aspect is important due to that it would mean a different treatment of the random variables in the forthcoming sections.

In order to proof the independence of this two variables we need to obtain a null correlation coefficient from an Independence Test (Ya-Lun Chou 72, Doménech 75). Nevertheless, this test has an initial requirement about the distribution of the random variables: they must follow a Normal distribution. Again, this condition implies two more tasks: first, we need to give admissible values —in relation to the sample— as parameters of the distribution; and second, we have to guarantee that the sample actually follows such a normal distribution.

We used the Kolmogorov Normality Test to verify that the experimental sample indeed follows a normal distribution both in the direction of the trajectory and in the direction perpendicular to the trajectory. We have also tested that both distributions are independent.

Empirical Distribution and Theoretical Normal Distribution

Considering the second column of the sample Table 2.1, we can calculate the empirical mean and variance and use them as estimators for the parameters of the theoretical Normal distribution N(μ,σ).

The mean estimators for x and y are:

$$\bar{x}_{21} = \frac{1}{21}\sum_{i=1}^{21} x_i = 0.6976 \text{ and } \quad \bar{y}_{21} = \frac{1}{21}\sum_{i=1}^{21} y_i = 0.5428$$

and the sample variances are:

$$\sigma_X = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} = 7.3122 \quad \text{and} \quad \sigma_y = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n-1}} = 1.7045$$

The μ and σ values for the theoretical distribution are chosen so that they are similar to the empirical sample but without laying outside a given confidence interval. We use a T-Student with n-1=20 degrees of freedom and a significance level α=0.05 (i.e., confidence of 95%) to obtain the confidence interval of the means. The critical value that appears in the table of such distribution is k=2.086 and the limits of the interval are computed using the following formula:

$$I_\mu = \bar{x} \pm k\frac{\sigma}{\sqrt{n-1}} \quad \text{and} \quad I_\mu = \bar{y} \pm k\frac{\sigma}{\sqrt{n-1}}$$

In that way, we have the x mean interval I_μ =[-2.713, 4.108] and the interval I_μ=[-0.252, 1.338] for the y mean. Since $0 \in I_\mu$ in both cases, then we can suppose null means for the Normal distributions of X and Y.

Analogously, when assigning a value for the standard deviation of the distribution, we compute the confidence interval using a ℵ-square distribution with n=20 and α=0.05. The values from the tables of such distribution are $\aleph^2_{(20,\ \alpha/2=0.025)}$ =34.17 and $\aleph^2_{(20,1-\alpha/2=0.975)}$ =9.59. And the upper and lower limits of the interval are computed using the equations:

$$L = \sqrt{(n-1)\frac{\sigma^2}{\aleph^2_{(n-1,\alpha/2)}}} \quad \text{and} \quad U = \sqrt{(n-1)\frac{\sigma^2}{\aleph^2_{(n-1,1-\alpha/2)}}}$$

And we have:

L_x=5.5942, L_y=1.3041 and U_x=10.5598, U_y=2.4616

As before, the deviation of the empirical distribution X (σ_X=7.3) must be included into the interval I=[5.5942, 10.5598] in order to take it as the theoretical deviation. And, since it is the case, we take σ_X=7.3. Equivalently, the theoretical deviation of *Y* is taken as σ_Y=1.7 because it is covered by the interval I=[1.3041, 2.4616].

Does the Empirical Distribution Fit the Theoretical Distribution?

Up to this point, we have specified two theoretical Normal distri butions: $N_X(0,7.3)$ and $N_Y(0,1.7)$. Nevertheless, we need to ensure that the sample – i.e., the empirical distri bution— actually fits these distri butions. We use the *Kolmogorov's Test for the fitness to a Normal distribution* (Cuadras 84) that can be used for small samples (where the number of values is smaller than 30). Briefly, this test consists on the computation of a statistic D_n that is defined as the maximum difference between an empirical distribution function S_n and the gi ven theoretical distri bution. If it is the case that the obtained D_n is smaller than the value in the Massey's Table (also in Cuadras 84) for the *n* number of elements in the sample, then we can affirm that the empirical distribution fits the theoretical one.

In order to compute the empirical distri bution we have first to organise the sample in increasing order $X_1<\ldots<X_i<\ldots<X_n$ and afterwards, to appl y the following formula:

$$S_n(x) = \begin{cases} 0 & \text{if } x < x_1 \\ 1/n & \text{if } x_i \le x < x_{i+1} \quad i = 1..n-1 \\ 1 & \text{if } x \ge x_n \end{cases}$$

In our sample *n* equals to 21, and the corresponding value from the Massey's Table is 0.289. The computed statistics for the *X* and *Y* distributions are D_n=0.115 and D_n=0.202 respectively. Since both values are smaller than 0.289, we can conclude that our empirical distributions fit the theoretical Normal distributions $N_X(0,7.3)$ and $N_Y(0,1.7)$.

Independence Test

Returning to our initial requirement for our statistical analysis, our goal is to test if the two distributions are independent –that is, if there is no correlation between the variables *X* and *Y*. In order to do that we have already tested the normality condition, and therefore, we can now apply the Independence test.

When $n \le 30$ the *t* statistic that is used to verify the hypothesis is distributed as a T-Student with *n*-2 degrees of freedom. More specifically, the *t* statistic has the following formula:

$$t = r \cdot \sqrt{\frac{n-2}{1-r^2}} \quad \text{where} \quad r = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sqrt{(\sum_{i=1}^{n} (x_i - \bar{x})^2) \cdot (\sum_{i=1}^{n} (y_i - \bar{y})^2)}}$$

In this formula, r is the sample correlation factor –also known as the correlation estimator of maximum likelihood— and we have r=0.4339 and t=1.882.

Using a significance level of α =0.05 for the T-Student distribution with 19 degrees of freedom, we obtain an interval (-2.101, 2.101) which includes the obtained t = 1.882. And this constitutes the proof of the independence of our random variables X and Y.

2.4.3 Error Modelling

Displacement Error

The previous subsection proofs that a sample of the ending positions of a 3m straight movements follow Normal distributions —$N_X(0,7.3)$ and $N_Y(0,1.7)$— that are associated to two independent random variables X and Y. This random variables correspond to the ending position co-ordinates.

In order to model the error of displacement that the sample reflects, we have obtained probability intervals over our Normal distributions such that they cover most of the sample. In that manner, considering the Standard Normal distribution N(0,1), the interval that is centred in the zero and that covers 95% of a sample has a total length of 3.92. In other words, if we have a sample following this N(0,1) distribution, the interval that covers the sample with a significance level of α=0.05 is (-1.96,1.96). Therefore, for a random variable X with such a distribution we have the following probability:

$$P[-1.96<X<1.96] = 0.05$$

Nevertheless, our N_X and N_Y distributions are not standard. This implies that we need multipl y their dispersions by 3.92 in order to obtain the intervals that cover 9 5% of our sample. The length of our interval for the X and Y variables are therefore 2 8.58 cm and 6.59 cm respectivel y.

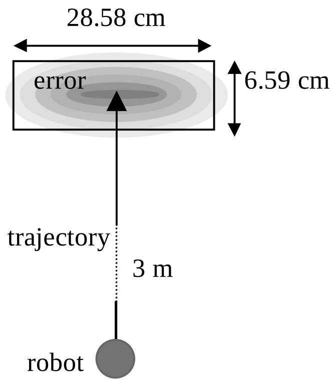


Figure 2.4: The position where the robot ends after a rectilinear displacement of 3 metres has this error rectangle associated.

Experimentally, we have observed that the rectilinear movement of the robots generate an elliptical distribution of the ending points —these are the positions where the robots finish after doing the same movements and starting from the same initial position. Furthermore, the distribution of the points has a higher density in the centre. Figure 2.4 shows schematically this idea. With our interval approximation we reflect this feature, the difference is that the ellipse is approximated by a rectangle in order to simplify the computation. Regarding the centred density, the Normal distributions with null deviations already reflect that.

These intervals will be used to associate a rectangle —representing the approximation of the error— to each position of the robot that belongs to a rectilinear displacement. In the case of the previous figure, what this rectangle basically means is that after 3 metres of displacement in straight movement, there is a high probability of the robot being actually inside the given rectangle. As in the figure, the rectangle width is in the direction of the trajectory and the length goes in the direction perpendicular to the trajectory.

All the analysis done up to this point only consider the first kind of movement presented at Table 2.1. The data related to the second movement has been studied in the same way. Both movements are basically the same —that is, a rectilinear displacement— the only difference is the distance covered by the robot: the second movement corresponds to a 6-metre displacement, twice the distance of the first one. The reason to repeat the same movement just varying the covered distance is that this will allow us to compare the data in order to conclude if the error grows proportionally to the distance or not.

The following Table 2.2 presents a comparison of the empirical mean and standard deviation of the *X* and *Y* distributions for both kinds of rectilinear movements.

	3 m	**6 m**
$\overline{x}$	0.33 cm	1.76 cm
$\overline{y}$	0.44 cm	1.13 cm
σ_X	7.29 cm	12.46 cm
σ_Y	1.68 cm	2.48 cm

Table 2.2: Empirical mean and standard deviation of the *X* and *Y* distributions of two rectilinear movements: 3 and 6 metre displacements.

Taking into account the fact that the confidence interval (see page 30) gives a margin for the difference between the deviations σ_X and σ_Y, we can assume that the ones for the 6 metre displacement doubles the deviations corresponding to 3 meters. Hence, we assume that the length of the error interval increases proportionally to the covered distance.

Turning Error

The remaining kind of movements from Table 2.1 at page 29 that have not yet being studied are the ones related to turns. The robots can perform 45-degree turns in both directions —right and left— without changing their position. After the turn, the robots were programmed to follow a 290 cm-long rectilinear trajectory. As previously, we compute the mean and the standard deviation of the sample. From the results in Table 2.3 we can observe that the sample deviation does not seem to be really affected by the turn.

	45° right turn	**45° left turn**
$\overline{x}$	-9.52 cm	-10.36 cm
$\overline{y}$	-1.66 cm	-1.49 cm
σ_X	5.51 cm	4.88 cm
σ_Y	2.56 cm	2.65 cm

Table 2.3: Sample mean and standard deviation of the ±45º turning movement (with a sequel displacement).

Nevertheless, the mean results for the two types of turns are surprisingly similar: in both movements the robot stopped at a mean position that was on the left and before the expected position (the robot ended at about 10 cm on the left and 1.5 cm before). This deviation is equivalent to a divergence respect the turning angle of 2 degrees on the left (more accurately, it corresponds to a difference of 1.89º for the 45º right turn and 2.06º for the 45º left turn).

From these observations the turning error has been modelled as a constant deviation of 2 degrees on the left. Therefore if, for example, a robot decides to turn 45º to the left, then we know that it is turning 47º on the average (and a theoretical 45º right turn will become on the average approximately a 43º turn).

The allowed turns for the robots have been restricted to multiples of 45º (that is, $45 \cdot k$ degrees). This means that the module of the robot that is in charge of performing the turns just repeats the same order k times. This implies that in order to compute angle deviations on the robots we only need to consider multiples of 2.

2.4.4 Error Propagation

In free space, a trajectory is composed of a set of alternating straight runs and turns. Given the error rectangle at the initial point of a trajectory, we want to determine the error rectangle at the end of each straight run. In this subsection we show how we increment the error of trajectory segments taking into account previous movements and accumulated errors.

Finally, during its navigation the robots detect and follow walls. Wall positions are computed from the location of the robots that detect them. Thus, those errors associated to the walls must be somehow similar to the accumulated trajectory errors. The end of this chapter explains the computation of the error associated to detected walls.

Error Associated to the Trajectory

Until now, we have seen that given a known initial position of the robot and given a rectilinear displacement, we can ensure —with a high probability— that its ending position will belong to the area of an error rectangle. What we have to specify now is the way in which the error rectangle should be expanded, starting from a given error rectangle and a robot movement.

The computation of the accumulated error could be thought as if each of the points in the area of the previous error rectangle followed the same movement than the robot. In that manner each of this points would accumulate the error that the current movement generates. This partial error rectangle would have the width in the same direction than the current displacement (and therefore the length would be perpendicular). Thus, the total error rectangle would be the one including all these partial rectangles.

We do not need to compute partial error rectangles for all the points in a previous rectangle. It is enough with just considering the four points at the corners.

Initially, we have an error rectangle *R* associated to the current robot position. We also know the vector of the previous robot direction $\vec{d}$ (this vector ends at *p*, the centre of *R*), the turning angle α and the length of the distance *l* covered after turning. The following Figure 2.5 illustrates the meaning of these initial values:

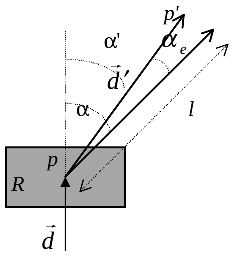


Figure 2.5: Initial data and computation of the new direction.

The following steps in pseudo-code give the idea about the computation of the increasing error:

1) Compute the real turning angle: $\alpha' = \alpha + \alpha_e$ where α_e is the deviation angle ($\alpha_e = 2 \cdot k$ and *k* comes from $\alpha = 45 \cdot k$).
2) Compute $\vec{d}$': the new displacement vector starting at *p*, with length *l* and with orientation the one that results from a rotation of α' degrees over the previous direction $\vec{d}$. Using the same notation, the point where $\vec{d}$' ends appears as *p*' in Figure 2.6.
3) Compute the new position of a copy of the rectangle error *R* so that its centre coincides with *p*' and its orientation does not change (that is, translate *R* from *p* to *p*').
4) Compute the error rectangle associated to the last straight movement of the robot: *r* is the partial error associated to $\vec{d}$' (notice that *r* is also centred at *p*').
5) For the four corners of the copy of *R* repeat (Figure 2.7 a):
   - make a copy of *r* and move it until its centre reaches the corner (do this without changing its orientation).
6) Compute a new rectangle *R*' such that includes all the copies of *r* and has the same orientation than *R*. (Figure 2.7 b)

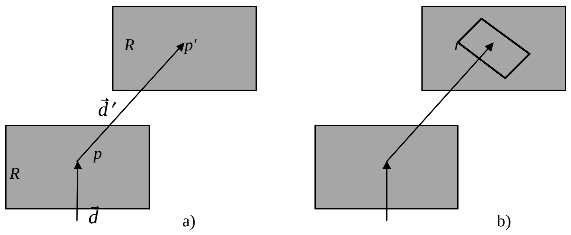


Figure 2.6: Illustration of the 2, 3, and 4 steps: compute $\vec{d}$ ', translate R along $\vec{d}$ ', and obtain *r*.

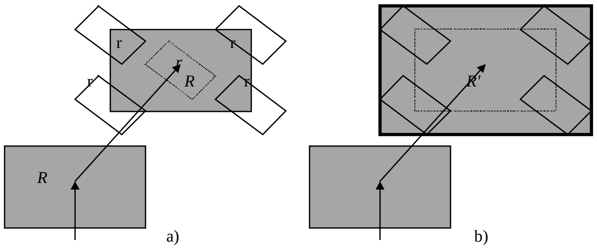


Figure 2.7: a) 5th step: copy *r* and move the copies to the corners of the copy of R b) Compute the resulting *R*'.

At the beginning of this chapter, it has been said that the robot trajectory is a sequence of turns and rectilinear displacements. This trajectory can be seen as a sequence of adjacent trajectory segments. Therefore, if we apply the previous algorithm to all the points in a trajectory we will have an error area growing along these segments. In fact, knowing the rectangle errors associated to the extremes of each segment, we can obtain the entire error area associated to the segment (notice that the final rectangle of a segment constitutes also the initial rectangle of its following segment). Figure 2.8 shows an example of the error propagation after a 45º right turn, a straight line, another 90º right turn and finally another straight line.

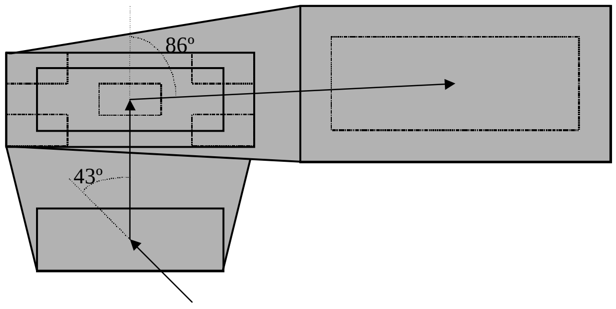


Figure 2.8: Error propagation for a trajectory with a 45º and 90º right turns

Error Associated to Wall Detection

The way in which the robots explore is by moving randomly in free space and following a wall when detected. The part of the trajectory that corresponds to the wall-following movement is a displacement that can be considered as being parallel to the wall. (This is a simplification of the real wall-following movement that consists in a zigzag that is driven by a differential servo system). The robot tries to keep constant its distance to the wall. This distance depends on the range of the infrared sensors for a specific wall material (see page 7). Each robot has associated its own average distance d of wall following as well as the corresponding average error e_d.

The portion of wall that is followed can be therefore modelled by a segment parallel to the corresponding trajectory segment. The wall segment is obtained from the trajectory segment —including robot's displacement direction and the segment length—, the distance d between the robot and the wall, and the side of detection (that is, if the robot detected the wall on its right or on its left).

The localisation error of the wall segment is computed considering the detection error e_d added to the corresponding trajectory segment error. This e_d is added in the direction of the robot's detection. Figure 2.9 a) depicts the generation of a wall segment that was detected on the right. Furthermore, Figure 2.9 b) shows a simplification of a). This simplification considers to be constant the error on the direction perpendicular to the movement. This constant value is set to be the average between the initial and the final error in this direction. The reason is the aim of reflecting the fact that the robot is somehow directed by the wall that it is following.

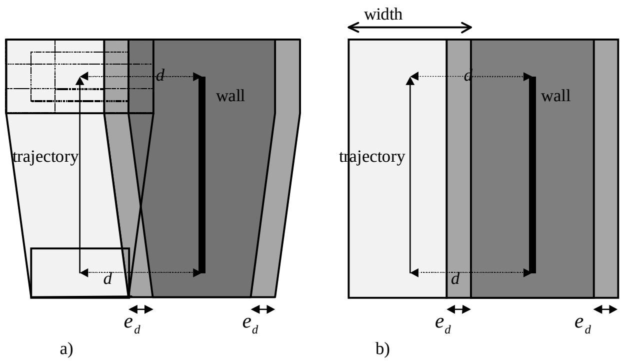


Figure 2.9: Computation of a wall segment from its corresponding trajectory segment: b) is the simplification of a).

Chapter 3

Fuzzy Techniques for Map Generation

The previous statistical error analysis is used to model the imprecise location of obstacles and walls by means of fuzzy techniques. Next section is a brief introduction to Fuzzy Logic and Fuzzy Sets —one of its central concepts. Their definitions give the necessary background for the subsequent Section 1.1, which specifies how we use fuzzy sets to represent the imprecision of detected map information. The third section describes a fuzzy logic-based algorithm that we have developed in order to fuse the obtained partial maps. This map information is in fact stored in the structure defined at the fourth section. The distribution of the map information has been designed so that implicit informatio n about environmental features such as corners or doorways can be specified. The Segment Completion section details the rules used to determine the existence and specification of such features. Finally, the last section gives a general description of the map generation process from a symbolic point of view.

3.1 Fuzzy Logic

Classical logic is a well-known model that can be used to formalise reasoning processes. However, classical logic can only be based on information that must be known to be certain. This is a very strong constraint that makes it useless for any application considering uncertain information. Fuzzy Logic is an alternative formalism that can deal with uncertainty (Zadeh 92).

Within the framework of classical approaches, Probability Theory is a widely established way of dealing with uncertainty. Nevertheless, the probabilistic framework depends on the availability of statistically supported data, which is often not available. Furthermore, in Probability, the lack of certainty about an event implies the certainty of its negation without leaving any space for ignorance (Godo 93). This goes against common situations, in which the poverty of the information suggests uncertainty about both the event and its negation. When building a map of an unknown environment, the initial value assignment has to reflect, for each cell, the ignorance about its occupancy. Probabilistic approaches assume equiprobability (i.e., $P_{\text{free}} = P_{\text{occupied}} = 0.5$) as a tacit agreement without any justification. However, equiprobability is a strong assumption because it implies that any cell has the same probability of being occupied than being free space. This hypothesis about initial probability values is equivalent to assuming that 50% of the space in the environment is occupied and that 50% is empty, and this is an assumption that is hardly fulfilled for most environments. Therefore, the main advantage of Possibility over Probability is that Probability cannot explicitly represent ignorance whereas Possibility is able to do it.

As we have said, most approaches using Probability for map generation do not justify the equiprobability assumption. Nevertheless, to the best of our knowledge, there is one author that comments this assumption. Yamauchi (Yamauchi 98) defines the initial Probability values in its grid map to be set to "the prior Probability of occupancy, which is a rough estimate of the overall Probability that any given location will be occupied". Nevertheless, this estimation is only empirically set without any formal justification. Getting into more detail, what he says is the following: "A value of 0.5 was used in all of the experiments described in this paper. In general this does not need to be an accurate estimate of the actual amount of occupied space. A prior Probability of 0.5 works fine in environments where only a small fraction of the total space is occupied, as well as in far more cluttered environments".

3.1.1 Fuzzy Sets

Let us consider a proposition of the form "X is A", where X is a variable and A is a vague predicate. X takes its values from an universe U and A denotes a Fuzzy Set of U that is represented by the following membership function:

$$\mu_A : U \rightarrow [0,1]$$

The notion of fuzzy set is an extension of the characteristic function from classical set theory. Considering a given context for a predicate and a universe, a fuzzy set represents this predicate attributing degrees of membership to the elements of the universe. Membership degrees belong to the [0,1] interval: 0 means no membership at all and 1 stands for full membership of the set.

The classical set operations of negation, intersection and union can be extended to fuzzy sets —represented here as A and B— as follows (Trillas 93):

- N: negation (for complementation)
  $\mu_{\neg A}(u) = N(\mu_A(u)) \quad \forall u \in U$ such that
  $N(0) = 1,\ N(1)=0,\ N(x) \geq N(y)$ *if* $x \leq y,\ N(N(x)) \geq x$
  Usually, $N(\mu_A(u))$ is taken as $\mu_{\neg A}(u) = 1-\mu_A(u)$
- T: t-norm (for intersection)
  $\mu_{A \cap B}(u) = T(\mu_A(u), \mu_B(u)) \quad \forall u \in U$ such that
  $T: [0,1] \times [0,1] \rightarrow [0,1]$
  $T(0,x) = 0,\ T(x,1) = x,\ T(x,y) = T(y,x),$
  $T(x,y) \leq T(z,w)$ *if* $x \leq z$ *and* $y \leq w,$
  $T(x,T(y,z)) = T(T(x,y),z)$
  Typical examples of t-norms:
  $T(x,y) = min(x,y),\ T(x, y) = x \cdot y$ and $T(x, y) = max(0,x+y-1)$
- S: t-conorm (for union)
  $\mu_{A \cup B}(u) = S(\mu_A(u), \mu_B(u)) \quad \forall u \in U$ such that
  $S: [0,1] \times [0,1] \rightarrow [0,1]$
  $S(1,x) = 1,\ S(x,0) = x,\ S(x,y) = S(y,x),$
  $S(x,y) \leq S(z,w)$ *if* $x \leq z$ *and* $y \leq w,$
  $S(x,S(y,z)) = S(S(x,y),z)$
  Typical examples of t-conorms:
  $S(x,y) = max(x,y)$, $S(x, y) = (x+y-x \cdot y)$ (known as probabilistic sum) and $S(x, y) = min(1,x+y)$ (bounded sum)

The relation of set inclusion is also extended to fuzzy sets:

$$A \subset B \text{ if and only if } \mu_A(u) \leq \mu_B(u) \quad \forall u \in U$$

3.2 Imprecise Location of Detected Wall Segments

The previous Chapter 2 explains how the robots explore an orthogonal unknown environment: the random movements that they perform in free space and the way they detect and follow walls. As a result of the exploration, each robot generates a partial map represented by a sequence of trajectory segments and the information necessary to compute the corresponding detected wall segments. In addition, Section 1.1 shows a statistical error analysis that estimates the error associated to both the trajectory segments and the wall segments. In the sequel, we will see how this statistical error analysis is used to model the imprecise location of obstacles and walls in the environment by means of fuzzy sets.

Our mapping process only considers environmental features with edges long enough to be followed by the robots. Small obstacles, such as chair or desk legs, are not represented because they generate spurious IR sensor readings that are considered as noise. In fact, this represents the main justification of our orthogonality assumption for indoor office-like environments. It is very frequent that this kind of environments present walls connected by right angles as well as office furniture —such as bookshelves or drawers— with rectangular shapes.

However, when a robot detects something and it follows its contour, the information that it stores is the same whether it corresponds to a wall or to the edge —or side— of an obstacle. Therefore, since most detected obstacles into our environments are actually walls, in the rest of this thesis we will refer to wall detection —and wall following— even for those cases where it may correspond to object detection —or object following.

3.2.1 Specification of our Fuzzy Set

The information that comes from the robots, used to represent the environment, is imprecise but not uncertain. It is imprecise in the sense that the co-ordinates of a detected wall segment do not necessarily correspond to the real location of the detected wall. This is because we have already seen that all detected points have associated errors. Regarding uncertainty, the information is not uncertain because the existence of a wall segment is guaranteed by the robot's wall following movement and by filtering spurious sensor readings.

Going back to the definition of fuzzy set in the previous 3.1.1 subsection, we can define a fuzzy set associated to each wall segment in the following way:

> Let us define a universe $U_l \subseteq \Re\times\Re$ of all locations in our environment. Then, given the information of a detected wall segment S, we define a fuzzy set F_s applying a vague predicate W "wall" over the values of $u_l \in U_l$. This fuzzy set is represented by a membership function:
>
> $$\mu_W : U_l \to [0,1]$$

This membership function is such that, if we take $\text{Area}_e \subseteq U_l$ as being the rectangle error computed for the wall segment S, then we have:

$$\begin{array}{ll} \mu_W(u_l) \in (0,1] & \textit{if } u_l \in \textit{Area}_e \\ \mu_W(u_l) = 0 & \textit{Otherwise} \end{array}$$

In other words, given a detected segment $S = [(x_1,y_1), (x_2,y_2)]$ and its error Area_e, we define its associated fuzzy set F_s such that it gives, for any point (x,y), its corresponding membership degree to a real wall. In fact, we take the function μ_W satisfying $\mu_W(x,y) = 1$ *iff* $(x,y) \in S$ and, from the value 1 at the points of S, μ_W decreases linearly until the value 0 at the borders of the error rectangle Area_e associated to S. In the case of the horizontal segment in the following Figure 3.1:

$$(x, y) \in \text{Area}_e \textit{ iff } \begin{cases} x_1 - e_1 \le x \le x_2 + e_2 \\ y - e \le y \le y + e \end{cases}$$

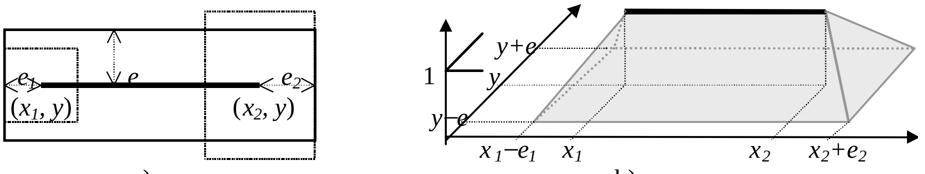


Figure 3.1: a) Representation of Area_e and S and b) representation of the imprecise segment (in grey, the associated fuzzy set F_s).

As Figure 3.1 a) shows, the length of this rectangle Area_e is $e_1+|S|+e_2$ being e_1 and e_2 the errors in the direction of displacement. As we have already said in the error analysis section 1.1, we consider the error e —in the direction orthogonal to the trajectory— being constant, and. this implies that the rectangle width is equal to $2e$. Furthermore, knowing that the error in the direction of displacement grows, we can infer that $e_1 < e_2$ — assuming that e_1 and e_2 are respectively associated to the initial and final points of S.

Notice that the previous figure shows an example of segment orientation —the horizontal with $y_1=y_2$— out of the two allowed by the orthogonality environment restriction (of course, the other allowed orientation is the vertical one with $x_1=x_2$).

3.2.2 Imprecise Segment Definition

At this point (and considering the notations that have been specified in the previous subsection), we are able to define our *imprecise segment* as the triple $S_I = (S, \text{Area}_e, F_s)$ where:

- S corresponds to a wall segment detected by a robot,
- Area_e is a location error rectangle that has been computed from S and from the accumulation of previous displacement errors, and
- F_s is an associated fuzzy set defining, for all the points in Area_e, their membership degree to a real wall (See Figure 3.1 b))

Actually, what we know from this imprecise information is that there is a portion of a real wall segment $[(a, b), (c, b)]$ being a, b, c co-ordinate values satisfying $y-e < b < y+e$, $x_1+e_1 > a > x_1-e_1$ and $x_2-e_2 < c < x_2+e_2$ (and similarly for the orthogonal case).

Finally, we represent —for simplicity— an imprecise segment in the following way:

$$S_I = [(x_1,y_1), (x_2,y_2), e, e_1, e_2]$$

3.3 Imprecise Segment Fusion

The fuzzy set F_s in the imprecise segment S_I determines the relation between a real wall in the environment and the points within and around the detected wall segment. Nevertheless, since the computer host collects all the information coming from the robots' exploration, the representation of the environment can become redundant and too fragmented. As a solution, we propose an approach that groups into one single segment those imprecise segments corresponding to pieces of the same wall. This section explains how two imprecise segments coming from detections of the same wall are merged —fused— in a new segment without loosing relevant information.

3.3.1 Conditions

First of all, we define the requirements that two imprecise segments S_I *and* S'_I must fulfil in order to take the decision of fusing them:

1. Equal orientation,
2. Error rectangles intersection, and
3. Close relative position.

More concretely:

- Both must have the same orientation. We consider that walls have only one orientation, so that a corner will always be considered as the junction of two different walls (see Figure 3.2).

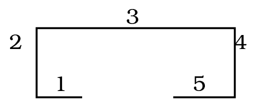


Figure 3.2: Example of the definition of 5 different walls.

- Their associated error rectangles must intersect. Two imprecise segments can be considered as portions of the same real wall only if it is plausible that their real positions determine a segment of the same wall (see Figure 3.3).

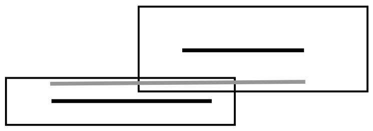


Figure 3.3: Two imprecise segments with intersecting error rectangles. The grey line represents a plausible position of the detected real wall.

- The relative distance between the detected segments S in S_I *and* S' in S'_I must be smaller than a certain threshold. It can be the case that the rectangle errors are so big that although the detected segments are very distant the rectangles still intersect. In order to control the maximum relative distance that will be allowed between segments that will be fused, this threshold can be changed in the application and acts as a 'grouping degree' parameter. The relative distance is determined by two measures Δx and Δy that must be smaller than the given threshold. These measures represent the minimum distance in the corresponding axis. For example, in the case of the two parallel vertical segments, Δx and Δy are computed in the following way (the horizontal case is equivalent):

$$S_I = [(x,y_1),(x,y_2),e,e_1,e_2]$$

$$S'_I = [(x',y'_1),(x',y'_2),e',e'_1,e'_2]$$

$$\Delta x(S_I,S'_I) = | x - x' |$$

$$\Delta y(S_I,S'_I) = \begin{cases} 0 & if(y_1 - y'_2)\cdot(y_2 - y') < 0 \\ min(| y_1 - y'_2 |,| y_2 - y'_1 |) & otherwise \end{cases}$$

In the Δy definition, the first part stands for the case in which both segments overlap. In that case the terms of the product have a different sign.

In this manner, Δx and Δy can be respectively viewed as measures of 'parallelism' and 'co-linearity' between *S and S'*.

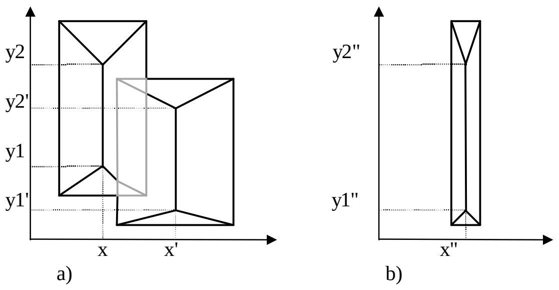


Figure 3.4: a) Two vertical segments that fulfil the conditions to be fused; b) Resulting segment fusion.

3.3.2 Fusion Computation

Once there are two imprecise segments —S_I and S_I'— that fulfil the fusion conditions, we define how to generate a new imprecise segment S_I'' from the combination of the previous ones. Following the example in Figure 3.4 we specify the vertical case. The horizontal case is obviously equivalent.

Intuitively, we specify the new vertical imprecise segment S_I'' = (S'', Area$_e''$, F_s'') where S'' includes the y and y' co-ordinates and has its x-co-ordinate x'' between x and x'. In addition, Area$_e''$ and F_s'' are associated to S'' and reflect that the combination of two pieces of information yields to a reinforced information —that is, the resulting imprecise segment is more accurate (see Figure 3.4).

In order to specify the combination of imprecise segments we have imposed the following intuitive premise:

- 'Those segments having small errors e should have more weight in the fusion than those with big errors'.

Still having in mind that we are in the vertical case —and therefore, we need to compute the values of x'' and e''— we consider the triangle projections of the fuzzy sets F_s and F_s' on the x-axis as Figure 3.5 shows.

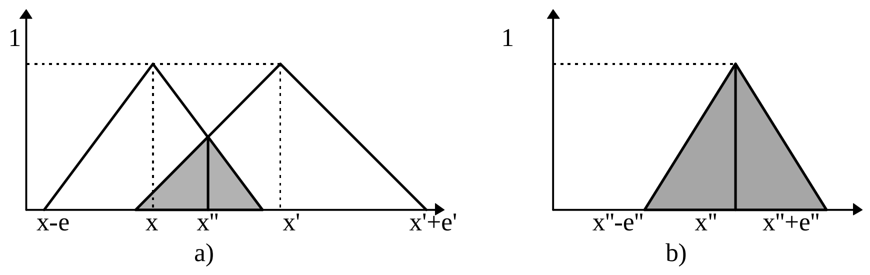


Figure 3.5: Triangle projections on the x-axis of a) the fuzzy sets F_s and F_s' and b) F_s".

Taking into account the conditions in the previous subsection, we define the mass of each triangle projection as being inversely proportional to their *e* error. Therefore, for each projection, the mass is 1/*e* —resp. 1/*e'*— and it is considered to be concentrated at *x* —resp. *x'*. In this manner, we can compute the centre of gravity of both masses as being the crossing point of the segments [(*x*,1),(*x*+*e*,0)] and [(*x'*–*e'*,0),(*x'*,1)]. The mass of the triangle projection of the fused fuzzy set *Fs*" is defined as the sum of the masses:

$$\frac{1}{e''} = \frac{1}{e} + \frac{1}{e'}$$

We are now able to specify the fusion of two imprecise segments:

$$S_I = [(x, y_1), (x, y_2), e, e_1, e_2]$$
$$S_I' = [(x', y_1'), (x', y_2'), e', e_1', e_2']$$

In order to obtain a new imprecise segment:

$$S_I'' = [(x'', y_1''), (x'', y_2''), e'', e_1'', e_2'']$$

By computing:

x" as the x component of the intersection point between the lines defined by [(*x*,1), (*x*+ *e*, 0)] and [(*x'*– *e'*, 0), (*x'*, 1)]:

$$x'' = x + \mathrm{r} \cdot (x + e - x) = x + e \cdot r$$

$$r = \frac{(1-0)\cdot(x' - (x' - e')) - (\mathrm{x} - (x' - e'))\cdot(1-0)}{((x+e) - x)\cdot(1-0) - (0-1)\cdot(x' - (x' - e'))} = \frac{x' - x}{e + e'}$$

$$x'' = x + \frac{e \cdot (x' - x)}{e + e'}$$

$$y_1'' = min(y_1, y_1'),\; y_2'' = max(y_2, y_2')$$

$$e'' = \frac{1}{1/e + 1/e'} = \frac{e \cdot e'}{e + e'}$$

$$e_1'' = \begin{cases} e_1 & if \;\; min(y_1, y_1') = y_1 \\ e_1' & otherwise \end{cases}$$

$$e_2'' = \begin{cases} e_2 & if \ \ max(y_2, y_2') = y_2 \\ e_2' & otherwise \end{cases}$$

Fusion Properties

As we have said, the Fusion is an operation F over pairs of imprecise segments (S_I, S'_I) that fulfil conditions of relative position. Fusion gives as result another imprecise segment:

$$S_I'' = F(S_I, S_I')$$

The components of the fused imprecise segment are computed by means of commutative operations:

- The x co-ordinate of the intersecting point of two lines do not depend on the order of consideration of the lines:

$$F_x(S_I, S_I') = x'' = x + \frac{e \cdot (x' - x)}{e + e'} = \frac{e \cdot x + e' \cdot x + e \cdot x' - e' \cdot x}{e + e'} = \frac{e' \cdot x + e \cdot x'}{e + e'}$$

$$F_x(S_I', S_I) = x'' = x' + \frac{e' \cdot (x - x')}{e' + e} = \frac{e \cdot x' + e' \cdot x}{e + e'} = F_x(S_I, S_I')$$

- The computation of the error e'' is also commutative:

$$F_e(S_I, S_I') = e'' = \frac{e \cdot e'}{e + e'} = \frac{e' \cdot e}{e' + e} = F_e(S_I', S_I)$$

- The remaining components depend on the minimum, which is, of course, a commutative operator.
- The conditions that S_I and S'_I must fulfil in order to be fused correspond to relative positions —that is, same orientation, error area overlapping and relative distance— are independent from the imprecise segment order of consideration.

Therefore, we can conclude that the fusion is a commutative operation:

$$S_I'' = F(S_I, S_I') = F(S_I', S_I)$$

In addition to commutativity, we can study some characteristics of the fusion from the equations of the computation of the components. In that manner, we analyse the horizontal case with $y_1 < y_2$ (the rest of the cases are equivalent) in order to comment these two characteristics:

1) It is possible to obtain an imprecise segment S''_I with the same coordinates than an imprecise segment S_I :

$$S_I'' = F(S_I, S_I^{id}) \text{ such that } \begin{cases} x'' = x \\ y_1'' = y_1 \\ y_2'' = y_2 \end{cases}$$

This is done by fusing *SI* with a segment S_I^{id} having the following co-ordinates:

$$x^{id} = x, \quad y_1^{id} \geq y_1, \quad y_2^{id} \leq y_2$$

2) Although it is possible to obtain an imprecise segment with the same co-ordinates —and thus, the corresponding errors *e1*, *e2*—, it is not possible to obtain the same error *e*. This is because:

$$e'' = \frac{e \cdot e'}{e + e'} = e \quad iff \ e = 0$$

And this is not possible because in our case *e* > 0 for all segments in our application (since the error in the robot's initial position is a matter of precision, *e* can be in any case considered to be bigger than 0).

In fact, we have the following inequalities:

$$e > e'' = \frac{e \cdot e'}{e + e'} \text{ because } e^2 > 0$$

$$e' > e'' = \frac{e \cdot e'}{e + e'} \text{ because } e'^2 > 0$$

Therefore, we have that the error of a fused imprecise segment is always smaller than the errors of the segments it comes from. This is intuitive because it gives support to the idea of that the uncertainty about the position of the real wall must decrease when there are two detections of the same wall.

Next figure shows two fusion examples that illustrate this result. Figure 3.6 a) is the fusion of two imprecise segments whose intersecting point coincides with the *x* co-ordinates of both segments:

$$x = 4, e = 4, \quad x' = 4, e' = 2, \quad x'' = 4, e'' = 1.33$$

Figure 3.6 b) represents the hypothetical situation of the fusion of an imprecise segment with itself. This last situation is unlikely to happen in the real application but it is useful to illustrate the characteristics of the fusion function:

$$x = x' = 3, e = e' = 3, \quad x'' = 3, e'' = 1.5$$

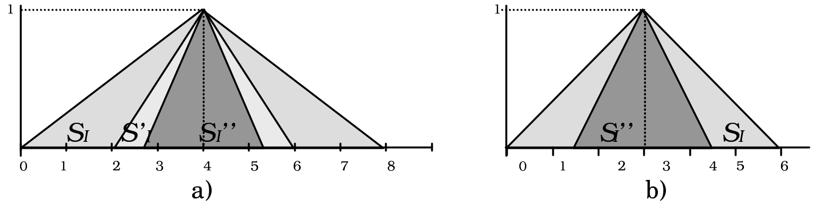


Figure 3.6: a) Fusion of two imprecise segments *SI''=F(SI,S'I)* b) Fusion of an imprecise segment with itself *SI''=F(SI,SI)*

Finally, we study the associative property of the fusion operation *F*. Again, we concentrate only on the computation of the *x* co-ordinate and its associated error *e* of three vertical imprecise segments *S_{I1}*, *S_{I2}* and *S_{I3}*:

$$F_x(S_{I1},S_{I2}) = x'' = x_1 + \frac{e_1 \cdot (x_2 - x_1)}{e_1 + e_2} = \frac{e_2 \cdot x_1 + e_1 \cdot x_2}{e_1 + e_2}$$

$$F_e(S_{I1},S_{I2}) = e'' = \frac{e_1 \cdot e_2}{e_1 + e_2}$$

$$F_X(F_x(S_{I1},S_{I2}),S_{I3}) = \frac{e_3 \cdot x'' + e'' \cdot x_3}{e'' + e_3} = \frac{e_2 \cdot e_3 \cdot x_1 + e_1 \cdot e_3 \cdot x_2 + e_1 \cdot e_2 \cdot x_3}{e_1 \cdot e_2 + e_1 \cdot e_3 + e_2 \cdot e_3}$$

$$F_e(F_e(S_{I1},S_{I2}),S_{I3}) = \frac{e'' \cdot e_3}{e'' + e_3} = \frac{e_1 \cdot e_2 \cdot e_3}{e_1 \cdot e_2 + e_1 \cdot e_3 + e_2 \cdot e_3}$$

$$F_x(S_{I2},S_{I3}) = x'' = \frac{e_3 \cdot x_2 + e_2 \cdot x_3}{e_2 + e_3} \quad F_e(S_{I2},S_{I3}) = e'' = \frac{e_2 \cdot e_3}{e_2 + e_3}$$

$$F_X(S_{I1},F_x(S_{I2},S_{I3})) = \frac{e'' \cdot x_1 + e_1 \cdot x''}{e_1 + e''} = \frac{e_2 \cdot e_3 \cdot x_1 + e_1 \cdot e_3 \cdot x_2 + e_1 \cdot e_2 \cdot x_3}{e_1 \cdot e_2 + e_1 \cdot e_3 + e_2 \cdot e_3}$$

$$F_e(S_{I1},F_e(S_{I2},S_{I3})) = \frac{e_1 \cdot e''}{e_1 + e''} = \frac{e_1 \cdot e_2 \cdot e_3}{e_1 \cdot e_2 + e_1 \cdot e_3 + e_2 \cdot e_3}$$

Therefore, we can conclude that the fusion operation *F* is associative:

$$F(F(S_{I1},S_{I2}),S_{I3}) = F(S_{I1},F(S_{I2},S_{I3}))$$

Next Figure 3.7 a) shows an example of three vertical imprecise segments S_{I1}, S_{I2} and S_{I3} with the following *x* and *e* values:

$$x_1 = 4\ ,\ e_1 = 4;\quad x_2 = 7\ ,\ e_2 = 2;\quad x_3 = 10\ ,\ e_3 = 3$$

The fusion computations are shown in the rest of the figure. The *x* co-ordinates and errors that are obtained in the subsequent fusion operations are, for each case:

b) $F_x(S_{I1},S_{I2}) = x'' = 6\ ,\ F_e(S_{I1},S_{I2}) = e'' = 1.\hat{3}$

c) $F_x(S''_I,S_{I3}) = x''' = 7.2\ ,\ F_e(S''_I,S_{I3}) = e''' = 0.9$

d) $F_x(S_{I2},S_{I3}) = x'' = 8.2,\ F_e(S_{I2},S_{I3}) = e'' = 1.2$

e) $F_x(S_{I1},S''_I) = x''' = 7.2\ ,\ F_e(S_{I1},S''_I) = e''' = 0.9$

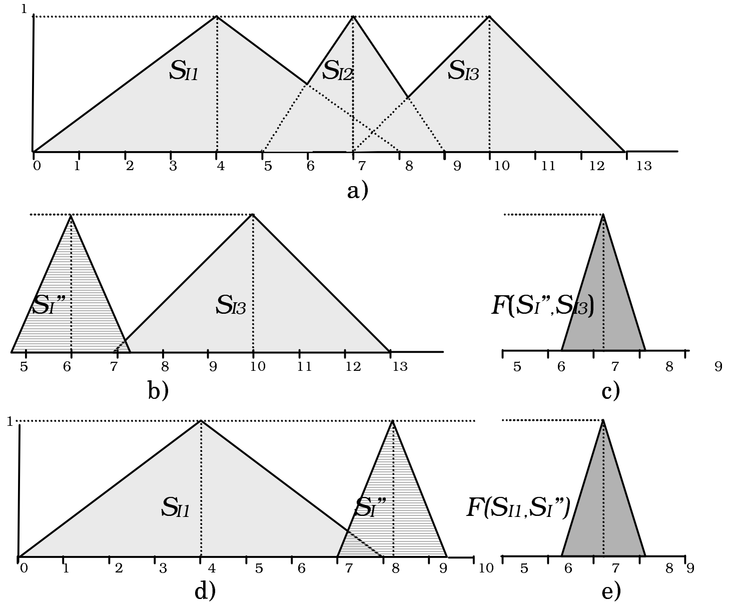


Figure 3.7: Example of fusion commutativity: a) three vertical imprecise segments S_{I1}, S_{I2} and S_{I3}; b) $S_I''=F(S_{I1}, S_{I2})$ and S_{I3}; c) $F(S_I'',S_{I3})$; d) S_{I1} and $S_I''=F(S_{I2}, S_{I3})$; e) $F(S_{I1},S_I'')=F(S_{I1}, S_{I2})$

After the demonstration of the fusion associativity, it would be natural to extend this property to the fused maps of our application. However, we cannot guarantee it because the fusion conditions are not always met. This is due to the change in the relative position after the fusion of the first two segments. In this manner, the resulting fused segment and the third segment may not always fulfil the fusion conditions, and therefore, the fusion between them cannot be computed.

Next Figure 3.8 illustrates this situation with an example of three vertical imprecise segments S_{I1}, S_{I2} and S_{I3} with the following x and e values:

$$x_1 = 2 \ , e_1 = 2; \quad x_2 = 3 \ , e_2 = 3; \quad x_3 = 7 \ , e_3 = 3$$

Figure 3.8 b) shows how the computation of the fusion of S_{I1} and S_{I2} gives as a result:

$$F_x(S_{I1},S_{I2}) = x'' = 2.4, \ F_e(S_{I1},S_{I2}) = e'' = 1.2$$

Graphically, it is straightforward to see that the fused segment does not overlap the remaining segment S_{I3}. Therefore, the next fusion $F(S_I'',S_{I3})$ cannot be computed. However, considering the segments in a different order allows the completion of the fusion process. The results are shown in Figure 3.8 d) and e):

$$F_x(S_{I2},S_{I3}) = x'' = 5,\ F_e(S_{I2},S_{I3}) = e'' = 1.5$$
$$F_x(S_{I1},S_I'') = x''' = 3.7,\ F_e(S_{I1},S_I'') = e''' = 0.9$$

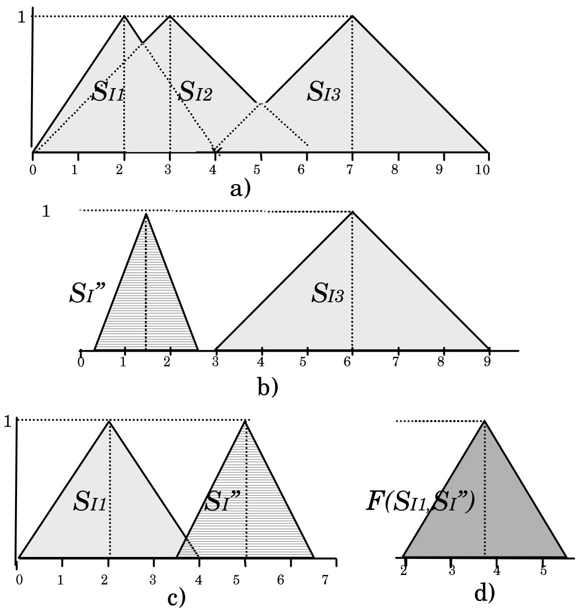


Figure 3.8: Although the fusion is associative, its conditions are not: a) three vertical imprecise segments S_{I1}, S_{I2} and S_{I3}; b) $S_I''=F(S_{I1}, S_{I2})$ and S_{I3} do not fulfil the fusion conditions; c) S_{I1} and $S_I''=F(S_{I2}, S_{I3})$; d) $F(S_{I1},S_I'')$

3.4 Global Map Representation

The map representation section 2.3.1 (page 26) explains that robots store their exploration trajectory in what we call *partial map*. This partial map contains sequences of trajectory segments and information about which ones correspond to wall following trajectories. In addition, if it is the case that a robot follows a wall and detects its end (it may be due to a corner or a doorframe), this ending position is stored as singular point. Both pieces of information —i.e., detection and singular points— are stored as labels of the trajectory segments.

Returning robots deliver their partial maps to the host computer. Partial maps are simple due to robots limited capabilities. Nevertheless, the host is capable of generating a much more complex representation of the environment: the global map. From the trajectory segments the host generates the corresponding imprecise trajectory segment by computing the associated error. The host also specifies the co-ordinates and errors of imprecise wall segments from those trajectory segments labelled as detections. These two kinds of imprecise segments are generated and stored in a dynamic list that grows as new partial map segments are added.

Once the host includes the incoming information, it groups this information and generates hypothesis about the existence of environmental features. As we have seen in the previous section, imprecise wall segments are grouped by their fusion, and in the following section will see how the completion process makes hypothesis about corners or doorways in the environment. Therefore, the host needs a representation complex enough to make such processing. For example, when adding an imprecise wall segment, it can have no more than one singular point so it looks reasonable to have a segment structure with a field for singular points. The problem is that afterwards, this imprecise wall segment can be fused with others and the limit for the number of singular points disappears. This forces to have an extra dynamic list for the singular points of each segment. Of course this could also be implemented by building a dynamic list for all the singular points in the global map that specifies, for each point, the segment to which it belongs. And, since to have different copies of the same information gives problems of contradictions when refreshing information, instead of having a copy of the segment, we will just have a pointer to the segment structure in the list of segments. The problem with this alternative distribution is that if we want to access directly the singular points of a segment we have to look over the whole list of singular points and this is not efficient. An intermediate solution is to have, on one hand, a dynamic list of singular points with pointers to their segments, and on the other hand, a dynamic list of pointers to singular points for each segment. This can look complex, but pointers have low memory cost and allow quick access to the information that facilitates its further treatment.

In addition to singular points, this symbolic approach represents two more features of the environment: corners and doorways. The global map has therefore a rather complex structure. Before giving a more detailed information, notice that Figure 3.9 shows an example that will help to illustrate the concepts. A global map contains 4 main dynamic lists with the following components:

- List of Segments: Ordered list that contains wall and trajectory imprecise segments. Each segment has the following structure:

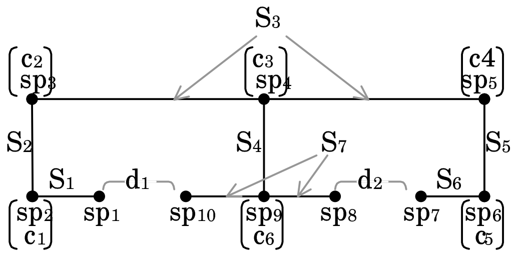
- Segment index (S_i): it is a unique identifier of the imprecise segment that gives its position in the list.
- Segment co-ordinates ($S = (x_1,y_2), (x_1,y_2)$): co-ordinates obtained from the robot partial map.
- Segment errors (e, e_1, e_2): errors associated to the segment, e goes in the perpendicular direction of the displacement and its value is used to order the list increasingly.
- Wall flag (Y/N): value that distinguishes if the segment represents a piece of detected wall —Y— or if corresponds to a robot trajectory —N.
- Pointer to its singular point list (→s_pt_l$_i$). Each segment has a secondary dynamic list of pointers to its singular points. This field is a pointer to that list.
- Pointer to its corner list (→c_l$_i$). A segment can also have any number of corners —only two L-corners but any number of T-corners— (we will see below what does this mean). This field is a pointer to a secondary dynamic list of pointers to corners.
- Pointers to its two doorways. Both extremes of a segment can end with the frame of a door. This is the maximum number of doorways that can be related to a segment because a wall segment is considered to be a continuous piece of wall.

- List of Singular Points: where each Singular Point is:
  - Singular Point index (sp$_i$): unique identifier.
  - Singular Point co-ordinates (x,y).
  - Pointers to its two segments ($\vec{s}_i, \vec{s}_j$): In general a singular point belongs to one segment, but further treatment can discover two perpendicular segments intersecting at a singular point that belongs to both segments.
- List of Corners: A corner is the intersection of a vertical and a horizontal imprecise wall segments. If the point of intersection is the extreme of both segments, the corner is considered to be of type L or L-shape. Otherwise, its type is T. The components of a corner structure are:
  - Corner index (c$_i$): unique identifier.
  - Corner type (L/T).
  - Pointers to its two segments ($\vec{s}_i, \vec{s}_j$).
- List of Doorways: A doorway is the gap between two collinear segments. It is supposed to have a known length that corresponds to the average size of an office-like environment door.
  - Doorway index (d$_i$): unique identifier.
  - Pointers to its two segments ($\vec{s}_i, \vec{s}_j$).

Of course, for the case of trajectory segments, most of the fields are empty. Nevertheless, as the next section will show, they are still useful for the completion process (which tries to fill the rest of structures).

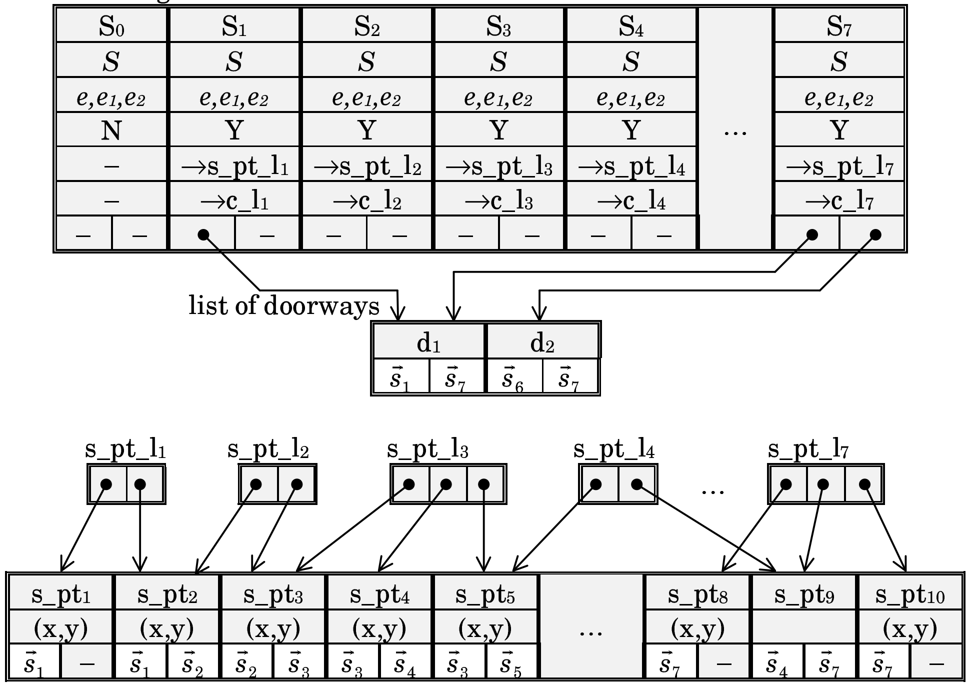
The following example shows an environment with its features and the map structure and information that the global map should eventually include in case the exploration were complete and without error. Nevertheless, some elements have been obviated with the aim of remarking representative elements. In this manner only an imprecise trajectory segment S_0 appears in the list of segments, and imprecise wall segments S_5 and S_6 are omitted because of their equivalence to S_2 and S_1 respectively.

Environment:



Global Map:

list of segments



S_0	S_1	S_2	S_3	S_4	…	S_7
S	*S*	*S*	*S*	*S*		*S*
e,e₁,e₂	*e,e₁,e₂*	*e,e₁,e₂*	*e,e₁,e₂*	*e,e₁,e₂*		*e,e₁,e₂*
N	Y	Y	Y	Y		Y
–	→s_pt_l1	→s_pt_l2	→s_pt_l3	→s_pt_l4		→s_pt_l7
–	→c_l1	→c_l2	→c_l3	→c_l4		→c_l7
– –	● –	– –	– –	– –		● ●

list of doorways

d_1		d_2	
$\vec{s}_1$	$\vec{s}_7$	$\vec{s}_6$	$\vec{s}_7$

s_pt1		s_pt2		s_pt3		s_pt4		s_pt5		…	s_pt8		s_pt9		s_pt10	
(x,y)		(x,y)		(x,y)		(x,y)		(x,y)			(x,y)				(x,y)	
$\vec{s}_1$	–	$\vec{s}_1$	$\vec{s}_2$	$\vec{s}_2$	$\vec{s}_3$	$\vec{s}_3$	$\vec{s}_4$	$\vec{s}_3$	$\vec{s}_5$		$\vec{s}_7$	–	$\vec{s}_4$	$\vec{s}_7$	$\vec{s}_7$	–

list of singular points

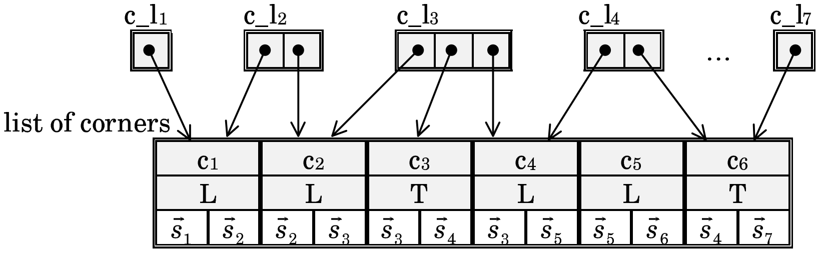

c_l1
c_l2
c_l3
c_l4
...
c_l7
list of corners
c1 L $\vec{s}_1$ $\vec{s}_2$
c2 L $\vec{s}_2$ $\vec{s}_3$
c3 T $\vec{s}_3$ $\vec{s}_4$
c4 L $\vec{s}_3$ $\vec{s}_5$
c5 L $\vec{s}_5$ $\vec{s}_6$
c6 T $\vec{s}_4$ $\vec{s}_7$

Figure 3.9: Example of representation of an environment with its features and the map structure and information that the global map should eventually include in case the exploration were complete and without error.

3.4.1 Fusion Implementation

Once we have seen how the global map represents the environmental information, we can specify how we fuse two imprecise wall segments without breaking references in the structure that could introduce inconsistencies in the information. These are the steps to follow:

1) Look for two segments S_I and $S_{I'}$ that fulfil the fusion conditions.
2) Follow the fusion computation formulae to generate a new imprecise segment $S_{I''}$.
3) Add $S_{I''}$ to the list of segments in decreasing order of their error e'' and without establishing any pointer.
4) Check if S_I or $S_{I'}$ have any pointer to a doorway or a corner, if it is the case:
    - copy these pointers in S_I and $S_{I'}$ to $S_{I''}$ so that they point to the same doorways and corners —d_i and c_i..
    - in these d_i and c_i, substitute the pointers to S_I or $S_{I'}$ by pointers to $S_{I''}$.
5) Repeat the fourth step for the singular points of S_I or $S_{I'}$. However, for each singular point s_pt$_i$ changed, its co-ordinates must be projected so that they belong to the new segment $S_{I''}$. Furthermore, before adding each s_pt$_i$ to $S_{I''}$, it must me compared to all previous s_pt$_{j,\, j\neq i}$ in $S_{I''}$, and if they are close enough to be considered the same, then s_pt$_i$ must be deleted. (This is if s_pt$_i$ is not related to a third segment, otherwise only the pointer to $S_{I''}$ will be deleted).

6) Delete S_I and S_I' from the list of segments (this includes a delete of their secondary lists of singular points and corners).

3.5 Segment Completion

The fusion of imprecise wall segments is a process based on segment information: orientation, co-ordinates and errors. However, it is also possible to use common sense knowledge in order to improve the global map and represent the environment more accurately. In our system, this common sense knowledge has been introduced by means of heuristic rules. Heuristic rules define, under certain conditions, the application of algorithms for different segment transformations and for environmental feature definitions. Segments can be extended or aligned, singular points can be added or redefined and doorways and corners can be defined into the global map structure.

Rule premises have two key aspects: the segment relative positions and the existence of trajectory segments going in between wall segments. On one hand, perpendicular segments with an intersection point close enough to both segments can lead to the definition of a corner. On the other hand, segments with the same orientation can be the clue for a doorway if they are collinear enough and if there is a trajectory segment going through their gap.

Regarding the consequent part or the rules, the given algorithms produce changes in the map, and this implies that the order of application of the heuristics is meaningful. We first check the rules that make segment changes without affecting the rest of the map and afterwards check those rules whose consequences affect other segments than the ones involved in the premises of the rules. Rules that define corners are of the former case, and doorway rules correspond to the later case. The following subsections get into more detail.

3.5.1 Corner Rules

A corner in the environment can be defined as the intersection of two perpendicular walls. As the previous section states, we define a structure in the global map called corner that is the intersection of a vertical and a horizontal imprecise wall segments. If the point of intersection is the extreme of both segments, the type of the corner is considered to be L — making reference to its shape. Otherwise, its type is T.

L-corners

To further illustrate the definition of L-corner, Figure 3.10 shows three different cases of pairs of segments that are candidates to form L-corners:

(S_1, S_2), (S_2, S_3) and (S_3, S_4). These three combinations have a relative distance —that is, the distance between the closer extremes— that is close enough to consider them as corners. On the contrary, the relative distance between S_4 and S_1 is bigger than a given threshold α that is a parameter of the implementation. Regarding the categorisation of the corners, the reason for labelling them with the L type is the situation of their intersecting point, which is located in the neighbourhood of the closer segment extremes.

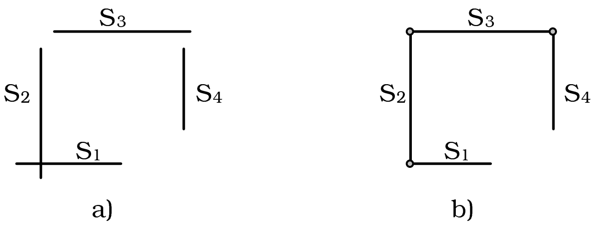


Figure 3.10: Three L-corners with different cases of related wall position: a) as they have been detected, b) after being defined.

Considering a wall segment pair (S_i, S_j) and their closer extremes $s_i \in S_i$ and $s_j \in S_j$, the following rule asks for several conditions in order to define L-corners in the map. First, the segments must be perpendicular. Second, the corner cannot possibly be closed if there is trajectory segments going between the closer extremes of the segments (s_i, s_j). Third, the intersection point p of the segments must be close enough to (s_i, s_j). And finally, if we are going to define a corner (p, S_i, S_j), this corner must be different to the rest of the corners in the global map corner list. The consequent of the rule is just the definition of the L-corner:

```
If     perpendicular($S_i$, $S_j$)  &
       $\nexists$ trajectory segment between ($s_i$, $s_j$)  &
       $\exists$ $p = S_i \cap S_j$ such that distance($p$, $s_i$)< $\alpha$ and distance($p$, $s_j$)< $\alpha$  &
       $\forall c_l$=($p_l$, $S_l$, $S_m$)$\in$corner list, $c_l \neq$ ($p$, $S_i$, $S_j$)
Then   - change the co-ordinates of $s_i$ and $s_j$ so that $s_i = s_j = p$  &
       - create a new L-corner ($p$, $S_i$, $S_j$) in the global map list of corners and update the
           corner-pointer-lists of $S_i$ and $S_j$  &
       - use $p$ to define a new singular point (pointing as well to $S_i$ and $S_j$) and update
           the singular-point-pointer-lists of $S_i$ and $S_j$.
```

When defining the new singular point, we use the co-ordinates of the intersection point p. Nevertheless, it can be the case that one —or both— segment already has a singular point in the extreme that is being treated.

In order to detect such a case, we must check if S_i or S_j have singular points s_pt_i and s_pt_j with the co-ordinates equal to s_i and s_j respectively. Due to the distribution of the data in our global map, this step must be done before changing the co-ordinates of s_i and s_j. Thus, if the answer to this check is positive, we have to update the s_pt_i and s_pt_j so that we end up with an unique singular point with co-ordinates p and pointing to S_i and S_j. (Of course, this last steps substitutes the one defining a new singular point). The results chapter will illustrate this case with more detail.

T-corners

Figure 3.11 gives three different examples of T-corner. The T-corners are defined by the segment intersection within the pairs (S_1, S_2), (S_1, S_3) and (S_1, S_4). In these cases the intersection point is close enough to both segments. Nevertheless, this is not the case of the segment pair (S_1, S_5), where the intersection point is further away from the S_1 segment than a distance α —the same parameter than the one used for L-corners. Notice that a corner is labelled as being of T kind whether it has the corresponding T shape —for example, (S_1, S_2)— or if it is cross-shaped — as the (S_1, S_3) corner.

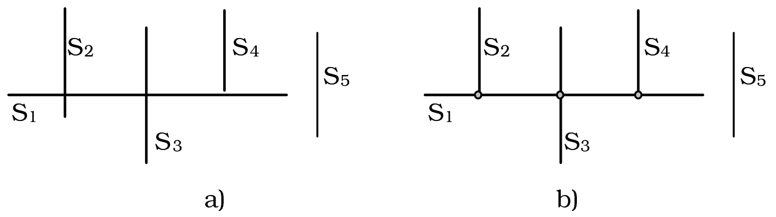


Figure 3.11: Three T-corners with different cases of related wall position: a) as they have been detected, b) after being defined.

Considering a wall segment pair (S_i, S_j), the family of rules used to identify and define T-corners in the map are similar to the one defined for L-corners. The consequent of these rules are analogous as well (including the definition of singular points). The difference is the measurement of the relative distance, which is measured between the intersection point p and each segment (this is a perpendicular distance between a point and a line and it is 0 if the point belongs to the line):

$$\underline{\text{If}} \quad \left\{ \begin{array}{l} \text{perpendicular}(S_i, S_j) \ \underline{\&} \\ \nexists \text{ trajectory segment between } (S_i, S_j) \ \underline{\&} \\ \exists\, p = S_i \cap S_j \text{ such that distance}(p, S_i) < \alpha \text{ and distance}(p, S_j) < \alpha \\ \forall c_l = (p_l, S_l, S_m) \in \text{corner list}, c_l \neq (p, S_i, S_j) \end{array} \right.$$

```
Then  - if p is close to one of the extremes of Si or Sj, then make the co-ordinates of this
          extreme equal to p  &
      - create a new T-corner in the global map list of corners (pointing to $S_i$ and $S_j$)
          and update the corner-pointer-lists of $S_i$ and $S_j$  &
      - use p to define a new singular point (pointing as well to $S_i$ and $S_j$) and update
          the singular-point-pointer-lists of $S_i$ and $S_j$.
```

3.5.2 Doorway Rules

A doorway is defined as the gap between two collinear wall segments. This gap is supposed to have an average door size that is introduced as a parameter β to the system. In practice, β is usually similar to α in the sense that both are related to the average door size, the difference is that β really represents the door size, whereas α refers to a distance used as threshold to define segment intersections —and whose value is usually taken similar to the β-value. As the previous corner rules, rules used to identify and generate doorways compare segments that fulfil some relative distance criteria. However, since the existence of doorways yields to environment information that can be afterwards used to move from one room to another, it becomes natural to ask for traversability requirements. This is translated into the premises of the rules as the condition of existence of a robot trajectory going through the gap.

The following rule is a generalisation of the ones used to identify and generate doors. They apply a collinearity test over a pair of segments (S_i, S_j) by checking that they have the same orientation and that their perpendicular distance is smaller than the α parameter. In addition, the distance between their closer extremes $s_i \in S_i$ and $s_j \in S_j$ be similar to an average size door β:

```
If    parallel($S_i$, $S_j$) and perpendicular_distance(S, $S_j$) < $\alpha$  &
      $\frac{1}{2}\beta \leq$ distance($s_i$, $s_j$) $\leq 2\beta$  &
      $\exists$ trajectory segment between ($S_i$, $S_j$)
      $\forall d_l$=($S_l$, $S_m$)$\in$doorway list, $d_l \neq$ ($S_i$, $S_j$)

Then  - align Si and Sj so that they become collinear  &
      - redefine $s_i$ and $s_j$ so that distance($s_i$, $s_j$) = $\beta$  &
      - create a new doorway in the global map list of doorways (pointing to $S_i$ and $S_j$)
          and update the doorway pointers of $S_i$ and $S_j$ &
      - define $s_i$ and $s_j$ as singular points of $S_i$ and $S_j$ and update the singular point
          pointer lists of $S_i$ and $S_j$.
```

Next Figure 3.12 shows an example of the definition of three doorways within the pairs (S_1,S_2), (S_2, S_3) and (S_4, S_5).

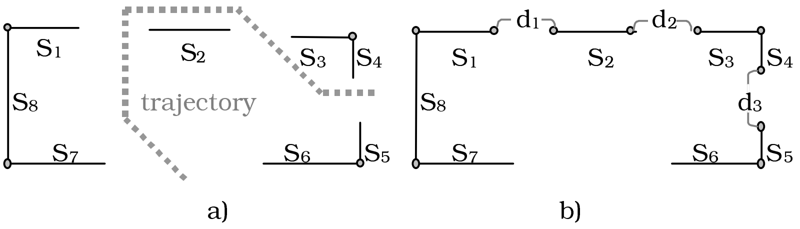


Figure 3.12: Definition of three doorways: d_1, d_2 and d_3: a) as they have been detected, b) after being defined.

When defining a new doorway from two wall segments there are two conditions that must be satisfied. First, the gap between the closer extremes of the segments must be of size β. And second, both segments must be collinear. Enlarging or shortening the segments is enough to satisfy the first condition. We just modify one of the segments, which is chosen on the basis of its length. In this manner, if the gap between both segments is smaller than β, then the longest segment is shortened. On the contrary, a gap bigger than β implies an extension of the shortest segment. (This is done with the aim of do not end up with very short segments).

Regarding the collinearity condition, it means a displacement of segments that may affect other map features. Doorway rules are applied after all corner rules with the aim of minimising their impact in the map. However, this does not avoid the problem completely and, therefore, this must be considered when choosing the segment to be modified and how this modification is going to affect the segment and its features. The chosen segment is the one with less impact propagation. That is, the segment with less characteristics that involve other segments. The way in which other segments are involved is also important. Thus, if for example, we change a segment S_i defining a corner with another segment S_j, we have to move S_i and, if necessary, redefine the extreme of S_j and the singular point at the corner. This is preferable than a case where S_i and S_j are related through a doorway, because in such a case, both segments must be displaced, and the propagation may continue. The previous Figure 3.12 shows an example where the definition of d_2 implies a displacement of S_2, which was already part of d_1. The change in S_2 not only implies a change in its singular points but a change of the previously defined doorway d_1. Fortunately, this kind of propagation ends in the corners (and the change of their singular points).

3.6 Map Generation

Previous sections of this chapter describe different processes that the host computer can perform over the environment information. These processes consist on the fusion and completion of imprecise wall segments in the global map. This section explains how these two processes are applied over the global map of the environment in order to transform its symbolic information into a more structured representation. The host computer builds and processes the global map incrementally, as it receives the partial maps from the exploratory robots.

The global map generation algorithm consists of three steps. The first one is the union of the current map in the host with the map perceived by a returning robot. This union is the union of both imprecise segment sets. The second step is the fusion of those imprecise wall segments that come from different detections but do correspond to the same wall. Finally, the algorithm completes the resulting map. The overall algorithm is as follows:

```
Map_Generation (RobotMap, CurrentMap)
begin
    NewMap = Fusion( CurrentMap ∪ RobotMap )
    return ( Completion(NewMap) )
end-map-generation
```

As we have said, this is an incremental approach, and it is so because of two main reasons. On one hand, not all troop members may return to give the information to the host. And, on the other hand, we want to have at any time the most plausible map based on the information obtained so far by the returning robots. Therefore, for all partial maps from new returning robots, the host computer updates the global map by executing once more this algorithm.

3.6.1 Map Fusion

The Fusion Implementation Section 3.4.1 defines the fusion algorithm that fuses two imprecise wall segments. The following fusion algorithm is applied to all segments in the global map —that is the union of the current global map with the new partial map. This algorithm is nothing more than an ordered call to the two-segment fusion function. The order in which segments are processed follows the same increasing error order than the list of segments in the global map M. Fusion is called with the global map M —which is copied to an auxiliary map M'— and the threshold fusion distance α (see L-corners Sect. at page 60) —which is copied to a constant ctt_dist. Fusion is as follows:

```
Fusion(M, ctt_dist)
begin
    M' = ∅
    while(M' ≠ M) repeat:
    begin
         for (all s∈M) do:
         begin
             min_dist = ctt_dist
             smin = ∅
             for (all s'∈M') do:
             begin
                   if (eq_orient(sl,s') & overlap_error(s,s') & distance(s,s')<min_dist)
                   begin
                       smin = s
                       min_dist = distance(s,s')
                   end-then
             end-for-s'
             if (smin = ∅)
             begin
                   add(s, M')
             end-then
             else
             begin
                   s" = fusion(s, smin )
                   add(s", M')
                   remove(smin, M')
             end-else
         end-for-s
         M = M',  M' = ∅
    end-wile
    return(M)
end-fusion
```

This algorithm consists of three nested loops: a while loop and two for loops. The nested for loops compare all segments in M with all segments in M'. Since M' is a copy of M with some fused segments, the maximum number of comparisons is n^2 —being n the number of segments in M— and corresponds to the case without fusions. The while loop is done until there are no more fusions to do over the map. The number of times this loop will be executed will depend on the distribution of the segments, but it wont be significant in any case. We can thus conclude that the complexity of this algorithm is quadratic to the number of segments in the global map: $O(n^2)$.

The general idea of this algorithm can also be depicted using the following written specification:

```
• For each step in a while loop:
    • treat all segments s in M following the order in the segment list of M
      —i.e., starting with the most precise segment. This treatment
      consist in a comparison of s with all segments s' in M', and then:
        • if there is any s' that can be fused to s —i.e., they have the
          same orientation, their associated error area overlap and
          their relative distance is smaller than α— we take the closest
          one (s_min) and generate a fused segment s" that is included
          into M' (s' is removed from M').
        • Otherwise, no segment in M' can be fused with s and,
          therefore, s is added to M'.
• After the treatment of all segments in M, we copy M into M', empty
  M' and keep doing the while loop until no more fusions are done, that
  is, while M ≠ M'.
```

The auxiliary map M' is useful to keep the order of segments during both the treatment process of segments in M as well as the map order while including new fused segments. Therefore, updating only the auxiliary map M' allows to sequentially treat all segments in M and to include new segments in M' consistently with the error order. The advantage of keeping the error order in the maps is that it ensures that the most precise segments are fused first. This implies a global reduction of the map imprecision. (Notice that the section 3.3.2 explains that the fusion of two imprecise wall segments leads to a new and more precise segment). However, the use of an additional map increases the complexity at the implementation level. The difference is that we are now treating segments —and their features— which come from two different data map structures.

We can use a small example to further illustrate how this algorithm works. Let M be a map with a list of four imprecise wall segments (s_1, s_2, s_3, s_4) as the ones in the following Figure 3.13:

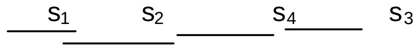


Figure 3.13: Example of the segments S associated to four imprecise wall segments

Then, the execution of the Fusion algorithm, considering as parameters M and α, is the following:

- Initially, M=(s_1, s_2, s_3, s_4) and M'=∅. We treat all segments in M:

$s_1 \in$M cannot be compared to any segment in M' because it is empty, therefore s_1 is added to M' and we have now M'=(s_1).

$s_2 \in M$ can be fused to $s_1 \in M'$, we thus add $s^{1,2}$ = fusion(s_2, s_1) to M' —in the appropriate place according to its error— and remove s_1 from M'. The resulting map is $M'=(s^{1,2})$.
$s_3 \in M$ cannot be fused to any segment$\in M'$ so, we just add s_3 to M', which is $M'=(s^{1,2},s_3)$.
$s_4 \in M$ can be fused to $s^{1,2} \in M'$ and to $s_3 \in M'$. Since s_4 is closer to $s^{1,2}$ than to s_3, we add $s^{1,2,4}$ = fusion(s_4, $s^{1,2}$) to M' having $M'=(s^{1,2,4}, s_3)$.

- For the next execution of the while loop $M=(s^{1,2,4}, s_3)$, $M'=\emptyset$. We do:
  (1) as before, $s^{1,2,4} \in M$ cannot be compared because M' is empty. We add the segment to the auxiliary map $M'=(s^{1,2,4})$.
  (2) $s_3 \in M$ can be fused to $s^{1,2,4} \in M'$. Since it is the only segment in the auxiliary map M', we first fuse them $s^{1,2,4,3}$=fusion(s_3,$s^{1,2,4}$) and substitute afterwards $s^{1,2,4}$ by $s^{1,2,4,3}$ in $M'=(s^{1,2,4,3})$.
- For the third execution of the while loop $M =(s^{1,2,4,3})$, $M'=\emptyset$,and we do:
  (1) $s^{1,2,4,3} \in M$ is the remaining segment. We end this third execution with the assignment $M'=(s^{1,2,4,3})$.
- Finally, the while loop ends because no more fusions are possible, that is, $M'= M =(s^{1,2,4,3})$.

3.6.2 Map Completion

Regarding the Completion process, we have already said that rules are applied in a specific order. The corresponding algorithm defines this order. It also uses the parameters α and β defined in the previous section. The algorithm is:

```
Completion (M, α, β)
begin
    change=0
    repeat
        M= apply_rules(M, L-corner-rules, α)
        M= apply_rules(M, T-corner-rules, α)
        M= apply_rules(M, Doorway-rules, α, β)
    while(change≠0)
    return (M)
end-completion
```

Chapter 4

Results

4.1 Robot Exploration Results

The robots used in this work have been designed and built by the Automatic Control department at the UPC (Technical University of Catalonia). When describing the robots' hardware characteristics at the second chapter (Section 1.1) three functional modules were specified. The navigation module —the one at the higher level— controls the robot's performance. This module cannot be tested until the mechanical part and both steering and perception modules are stable. Furthermore, this module was designed to be a piece of C code copied to a robot's EPROM. These two characteristics of the navigation module yield to the convenience of developing a simulator system. A simulator was built so that the navigation code could be directly portable to the real robot. Therefore, this simulator was built in order to develop —and test— a navigation module in parallel to the robot building process.

This simulator is able to represent different environments, it allows the user to choose the kind of robot that will explore an environment, it simulates the exploration (including the returning trajectory) and, finally, it shows the resulting partial map —which will be communicated to the host.

4.1.1 Environments

The simulator has a screen area dedicated to represent the map of the environment. Getting into more detail, this screen area is a rectangle comprising 256×512 pixels. Each pixel is meant to correspond to a square of

10cm^2 of the real environment. Therefore, it is able to represent any environment of size 25.6m by 51.2m. Since this work considers indoors orthogonal environments, the environments defined for the simulator represent 10cm width orthogonal walls. The gap between collinear walls represent doorways and they have a standard size of 80cm —that is, 8 screen pixels.

First, we have been using the environment that appears in the following Figure 4.1. In the sequel, we will refer to it as Env1. It represents two rooms communicated by two doorways. The biggest room contains an obstacle.

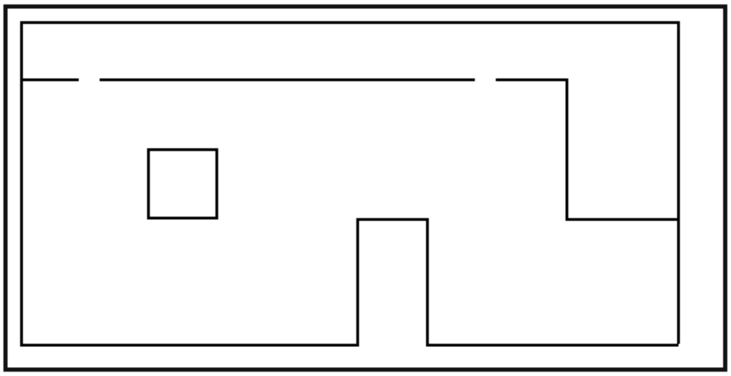


Figure 4.1: First environment (called Env1).

The simulator does not represent depth information, therefore, all the reachable areas in the map are supposed to be easily passable. As we have said, we consider indoors orthogonal environments. Usually, real environments with these characteristics are defined as being office-like. Nevertheless, our restriction is just orthogonality, and to show that we can deal with environments of different complexity, we also utilise the one in the next Figure 4.2:

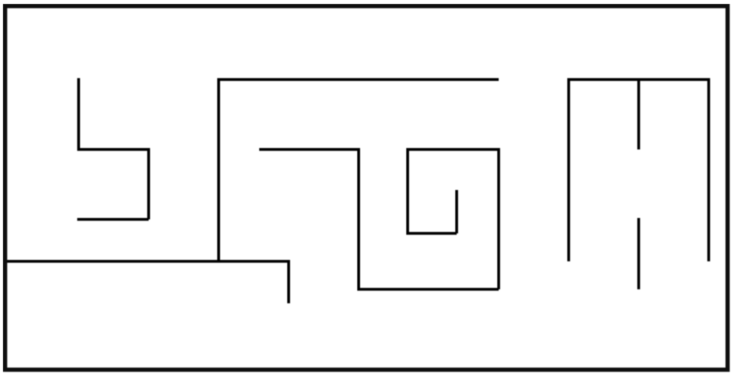


Figure 4.2: Env2 environment.

Finally, we use a third environment that is more realistic than the previous ones. It is called Env3 and has 9 rooms and two intersecting corridors. Again, the doorway gaps are 80 cm long (see Figure 4.3).

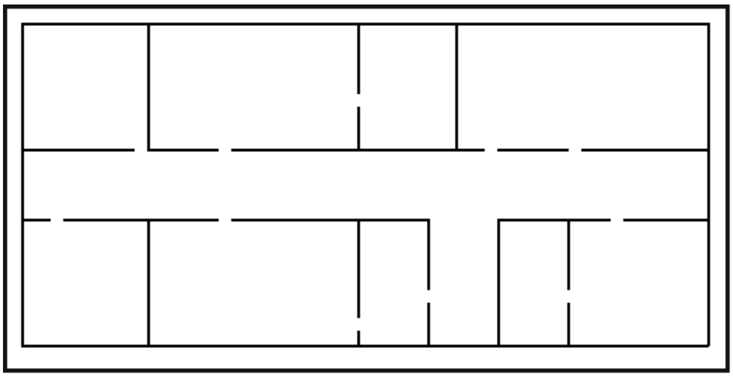


Figure 4.3: Env3 environment.

4.1.2 Robots Strategies

The Robot navigation Section 1.1 in the second chapter defines three different turning probabilities. They are associated to the robots in order to obtain three qualitatively different behaviours:

- “anxious”,
- “normal”, and
- “calm”.

Robots with different behaviours perform differently. The simulation of several robots’ explorations illustrates this idea. Anxious robots are defined by a high turning probability. This tends to generate intricate trajectories as the one in the following Figure 4.4. It corresponds to a simulation of an anxious robot that explores the Env1 environment. The starting position of the robot is fixed to (25,125) —in map screen co-ordinates. In general, since anxious robots turn so often, they tend to cover a small part of the environment, and therefore, they find a small number of walls. However, their focusing on a small area implies a good knowledge of it. This gives more information about free space than about wall detection information, but as we have already seen, this can also be useful.

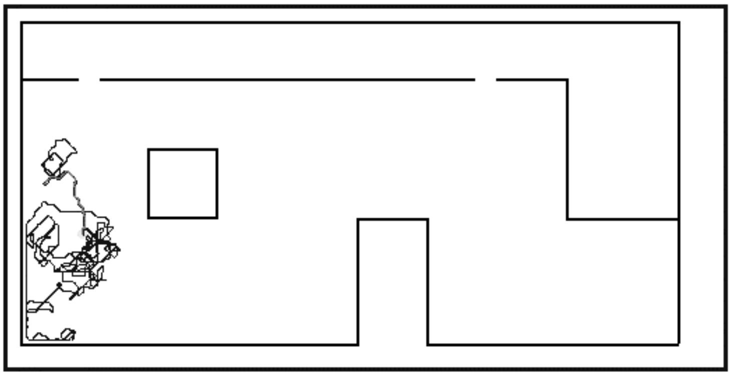


Figure 4.4: Exploration of the Env1 environment with an anxious robot with a significant tendency of turning to the left.

Nevertheless, anxious behaviours can sometimes give a significant amount of wall information. Next Figure 4.5 is an example:

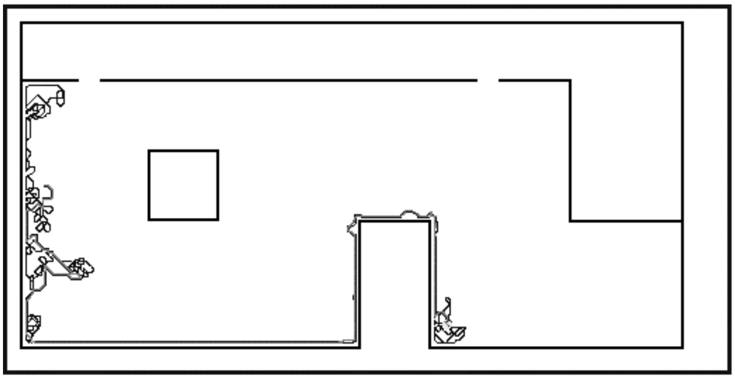


Figure 4.5: Exploration of the Env1 environment by a anxious robot with a significant tendency of turning to the right.

We can see in Figure 4.6 the performance of a normal robot over the same Env1 environment. In this case, the robot explores a wide area and finds walls located further away from its starting point. This is due to the value of the turning probability, which is small enough to allow relatively long displacement in free space —when nothing is detected.

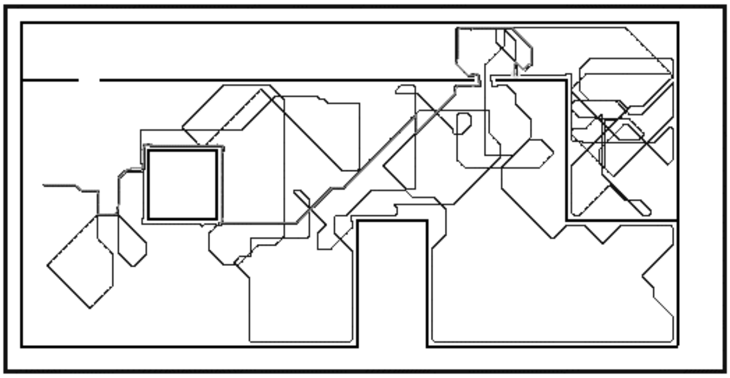


Figure 4.6: Env1 exploration of a normal robot with a right turning tendency.

Finally, we can see an example in Figure 4.7 of robot exploration when the robot has a small turning probability. As the one in the example, calm robots tend to perform long rectilinear displacements in obstacle-free areas.

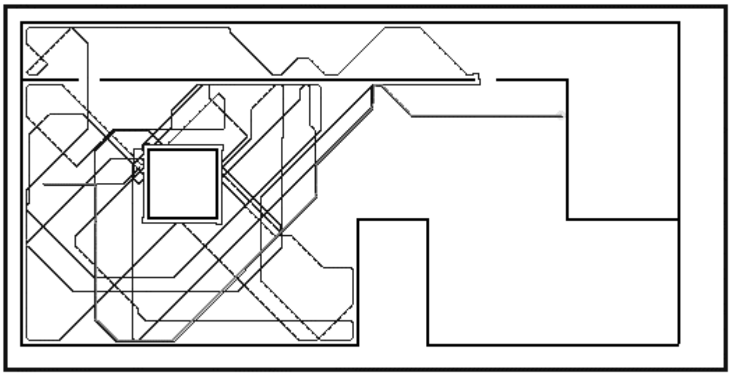


Figure 4.7: Simulation of the exploration of a robot —with left turning tendency and a calm behaviour— over the Env1 environment.

Up to this point we have seen the performance of the exploration of robots in the Env1 environment. Most decisions taken during robot navigation are highly randomised. The previous examples have shown us that the assignment of turning probability values imposes some behaviours. Nevertheless, there are other factors that define the actual navigation. On one hand, the layout of the environment, and one the other hand, the probability of turning right or left (defined as P_4 at Section 1.1). We will see some examples of exploration of the Env2 environment. As we have already said, Env2 is not necessarily a realistic environment, but it is useful to illustrate this idea.

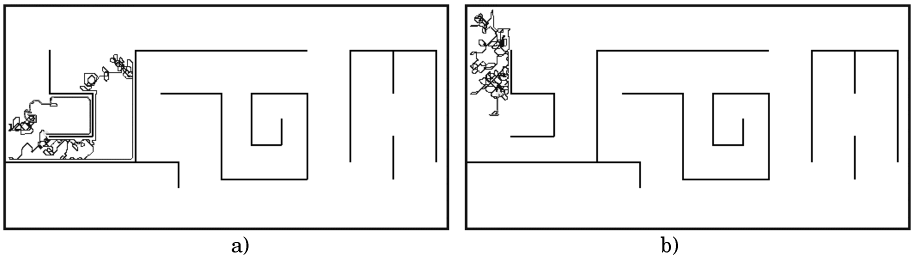


Figure 4.8: Exploration of Env2 by a a) right anxious robot —that is, with a right turning tendency and an anxious behaviour— and b) a left anxious robot.

The figure above (Figure 4.8) shows two cases of anxious robots. A priori, a robot that turns right more often than to the left does not necessarily perform better than a robot that behaves inversely. Nevertheless, we can see that under certain circumstances (which have an important random component) one kind of robots manages better into one environment than the other. The following Figure 4.9 gives two more examples of robots with a left turning tendency that have some difficulties in not becoming trapped in one area.

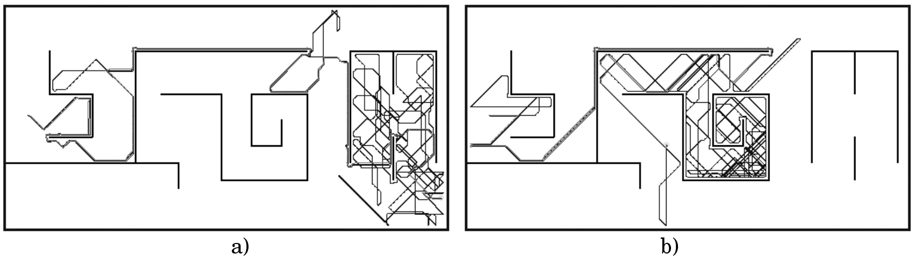


Figure 4.9: Exploration of Env2 with a) a normal robot and b) a calm robot, both with left turning tendencies.

This does not mean that left robots perform worse than right robots, it just justifies the necessity of including this new probability. Therefore, we can conclude that we obtain six different behaviours by combing the turning tendency probability with the three turning probabilities that define the anxious, normal and calm behaviours.

Robot Return

Due to hardware restrictions, the robots have been designed to store limited trajectories. Since a trajectory is defined as a sequence of robot turning positions, the simulated robot stops after doing a certain number of turns. From this position, the robot moves back towards its initial position. The trajectory that it follows consists in a simplification of its own trajectory. The simplification is computed by eliminating closed trajectory loops. All exploration figures have a yellow circle, which determines the returning point, and a red trajectory that indicates the returning path.

Odometry Error

We have already said that the robots and the simulator that is used here were developed in parallel. The error of the robots that was considered in this simulator does not come from the error study of the second chapter because this analysis has been done over the real robots (and when the simulator was built the robots were not finished). The error used in this simulation is a kind of general prediction of what could be described as a quality factor of the robot position information. This quality factor depends on the covered distance L as well as on the number of turns N. This dependency is inversely proportional according to the following formula:

$$F_Q = \frac{1}{(1 + L/k_L + N/k_N)}$$

Were k_L and k_N are constants that control how the quality factor F_Q decreases. Their values were set in the simulator to k_L=100 and k_N=30. They consider that L is given in meters and k_N corresponds to the constant of the quality of turns.

Partial Maps

After exploring, robots communicate their partial maps to the host computer. The structure of these maps has already been presented at the Partial Map Subsection 2.3.1. Thus, in this subsection we will just see some examples in order to add a few comments about environment coverage and data accuracy. As examples, we use the partial maps that were obtained from the explorations heretofore presented.

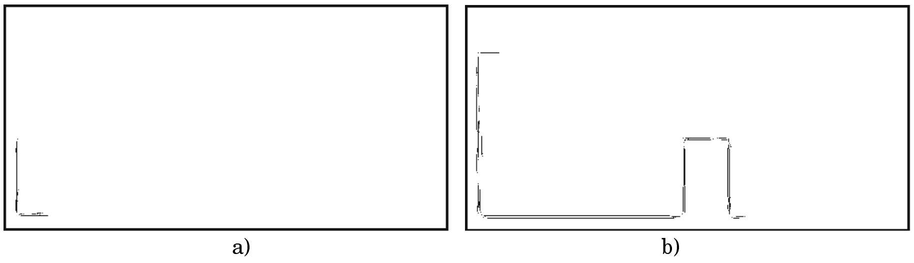


Figure 4.10: a) Partial map obtained from the exploration of a left anxious robot over the Env1 environment. It corresponds to the exploration shown in Figure 4.4 . b) corresponds to the exploration of a right anxious robot in the same Env1 environment (see Figure 4.5).

Regarding the exploration of the Env1 environment, the Figure 4.10 above confirms that anxious behaviours actually give information about walls and free space near the robot's initial position. Furthermore, the first partial map in the figure is very accurate because of two reasons. First, a robot stops after a number of turns, so anxious robots cover less distance than non-anxious ones —and therefore, the decreasing of its quality factor is smaller. And second, for this particular case, walls have been detected in a relatively early stage of the exploration trajectory —no much error was accumulated yet. This last reason is purely random and does not apply for the second partial map in the figure. In this case, the robot returns to its initial position detecting again the same bottom wall but with less accuracy.

Figure 4.11 shows the partial maps from the two remaining explorations of Env1. Both robots accumulate more error because they cover long distances —and wide areas. The decay of the quality factor can be clearly noticed in the detection of the obstacle in the partial map of the normal robot.

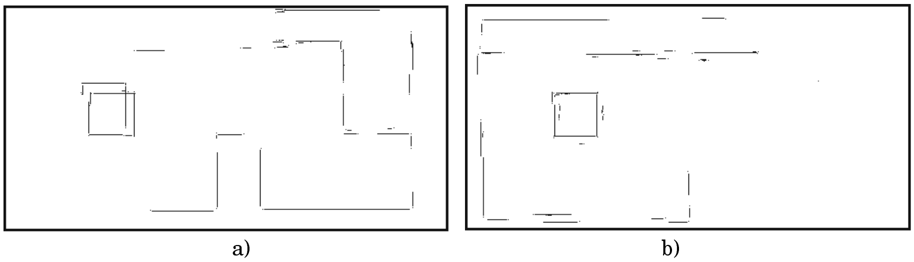


Figure 4.11: Partial maps of the exploration of Env1. a) partial map of the right normal robot in Figure 4.6, b) of the left calm robot (see Figure 4.7).

Finally, we see the partial maps corresponding to the Env2 environment. As we have already said, Env2 is particularly difficult for some robots that get trapped in an area. However, this is not necessarily bad, because if the area is different for several robots, we can obtain complementary coverage of the environment. Next Figure 4.12 illustrates this the four partial maps that correspond to Env2 exploration.

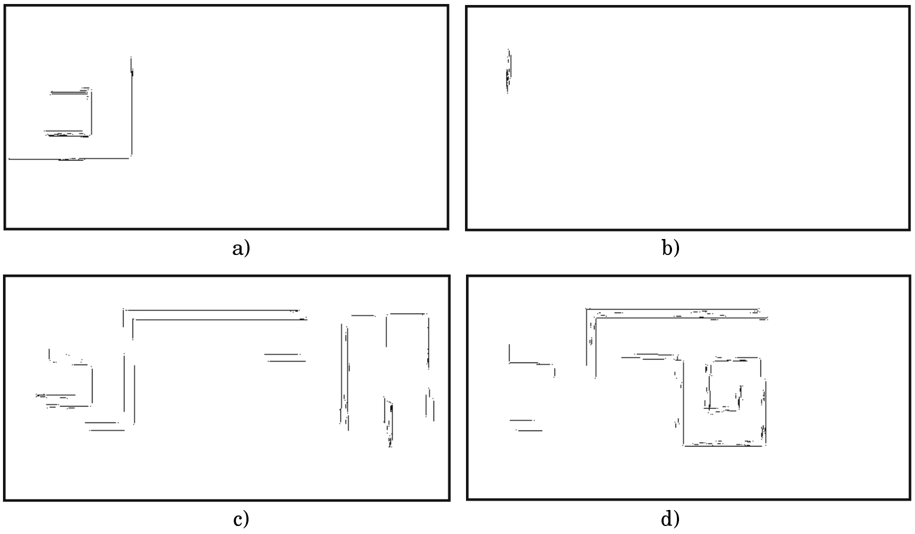


Figure 4.12: Env2 exploration: a) right anxious robot from Figure 4.8 a); b) left anxious robot (Figure 4.8 b)); c) left normal robot (Figure 4.9 a)); and d) left calm robot (Figure 4.8 b))

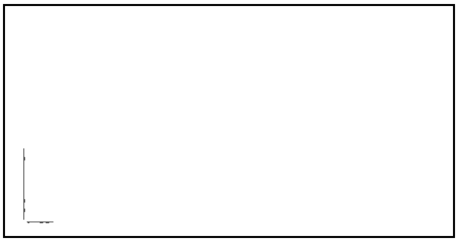
4.2 Map Generation Results

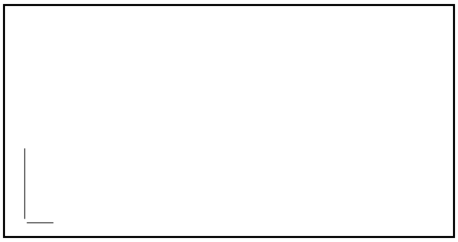
At the introduction of the second chapter (see page 21) we state that our approach has two main steps. The first is a distributed exploration and the second corresponds to the global map generation. Along the previous section of this chapter we have seen the kind of partial maps that the robots obtain from exploration. In this section, we concentrate in the map generation process. We show some results of the incremental fusion of new partial maps with the current global map at the host computer. Afterwards, we will also comment the map completion process.

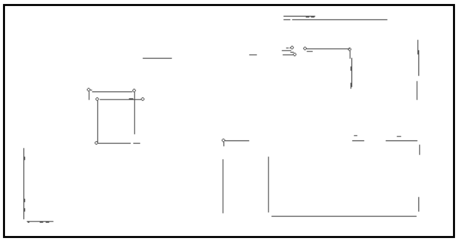
4.2.1 Fusion

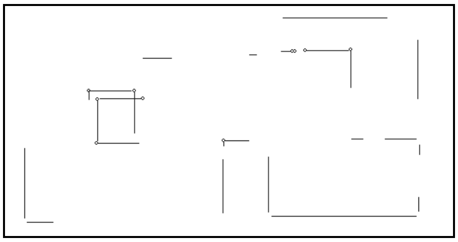
Last section of the third chapter (see Sect. 1.1) describes the general algorithm of Map generation. This algorithm first includes all segments from a newly delivered partial map to the current global map and calls a fusion function with the new current global map.

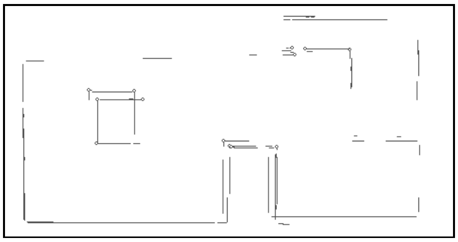
Next Figure 4.13 shows a sequence of the implementation of this step for the partial maps that correspond to the exploration of the Env1 environment. Maps shown on the left side of Figure 4.13 correspond to the union of partial maps that have being communicated to the host and maps on the right constitute the sequence of fused maps. In fact, the host does not compute the fusion of these communicated maps but a fusion of the union of the currently fused global map with the new incoming partial map. Maps in the figure have been presented in such a way in order to reflect the gain of the fusion process —which reduces the number of segments. The caption of the figure explains the order of communication of the partial maps after subsequent explorations of the Env1 environment.

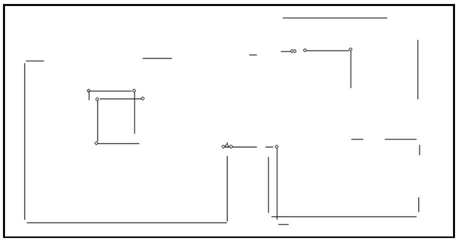

a)


b)


c)


d)


e)


f)

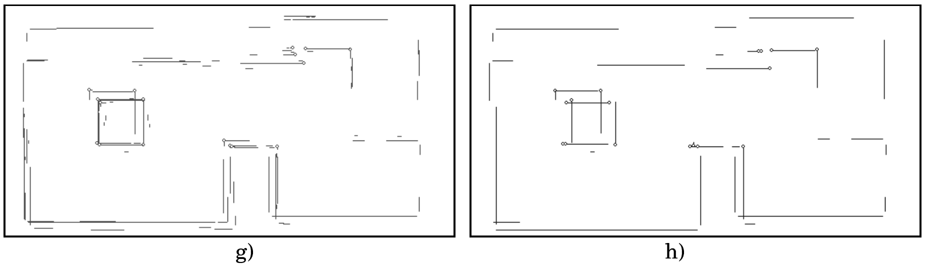


Figure 4.13: Map generation of the Env1 environment. a) The first robot that communicates its partial map is the left anxious (see Figure 4.10 a)); b) fusion of the current map; c) Shows the union of the previous partial map in a) with the second communicated map, which corresponds to the right normal robot exploration (see Figure 4.11 a)); d) Gives the fusion of the union of the fused map in b) with the new partial map in c). Afterwards, the right anxious robot communicates its partial map (see Figure 4.10 b)): e) Shows the union of previous partial maps and f) Is the fusion. Finally, the map from left calm robot (see Figure 4.11 b)) arrives to the host: in g) We have the union of all partial maps and in h) The final fusion.

Considering the previous Figure 4.13, there are several aspects to comment:

- Partial maps are filtered so that spurious readings are not considered. Comparing the partial map from a) and the one that came from exploration (see Figure 4.10), we can see that the map generation process filters some detected points (those segments having the same initial and final position).
- From the partial map information, the map generation process computes the corresponding imprecise segments (see Sect. 3.2.2). Imprecise segments are triples S_I=(S, Area$_e$, F_S) were S is the segment detection, Area$_e$ is the error area asociated to S and F_S is the corresponding fuzzy segment. All figures in this section have a segment representation that corresponds to the S component.
- Among the conditions that two segments must accomplish in order to be fused, there is one that can be tuned by the user of the map generation program at the host. This is the fusion threshold (see Sect.3.3.1). For the maps in the previous figure, his parameter has been set to 0.8m, which corresponds to the gap of a standard doorway. Therefore, the map generation process will not fuse collinear segments if their relative distance is bigger than 0.8m (even if their error areas Area$_e$ overlap).

Regarding this fusion threshold, the assignment of its value is clearly tight to the kind of environment. Considering now the less realistic Env2 environment, we can use a bigger threshold to fuse segments. Next Figure 4.14 shows how this has been done with a threshold of 1.2m.

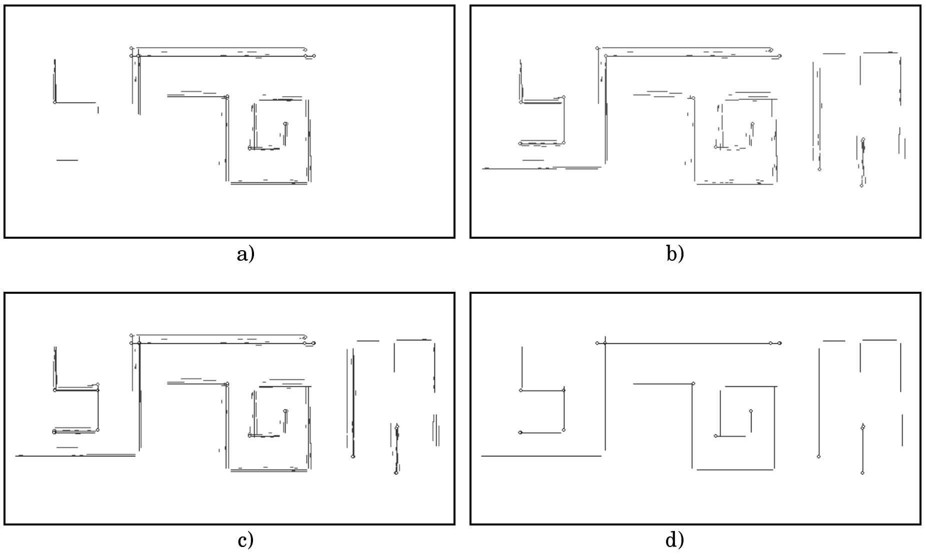
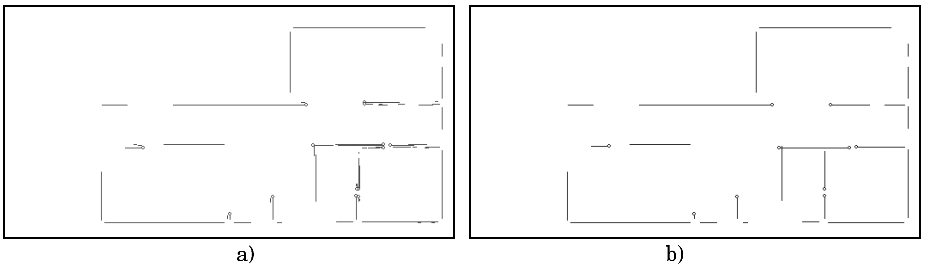


Figure 4.14: Map generation process of the Env2 environment. Communicated partial maps are in red, and the result of the fusion function is in blue. a) Two partial maps and the resulting fused map, partial maps came from the left anxious robot (see Figure 4.8 b)) and the left calm robot (Figure 4.9 b)); b) Four partial maps: the ones at a) plus the partial maps from the right anxious robot (Figure 4.8 a)) and the left normal robot (Figure 4.9 a)); c) Maps in b) with a superposition of their fused map; d) The fusion of the map shown in c).

Finally, we see one more example of fusion. In this case (Figure 4.15), we consider the Env3 environment, whose layout is more office-like than the previous environments. In this case the host generates the global map from the exploration of three robots.



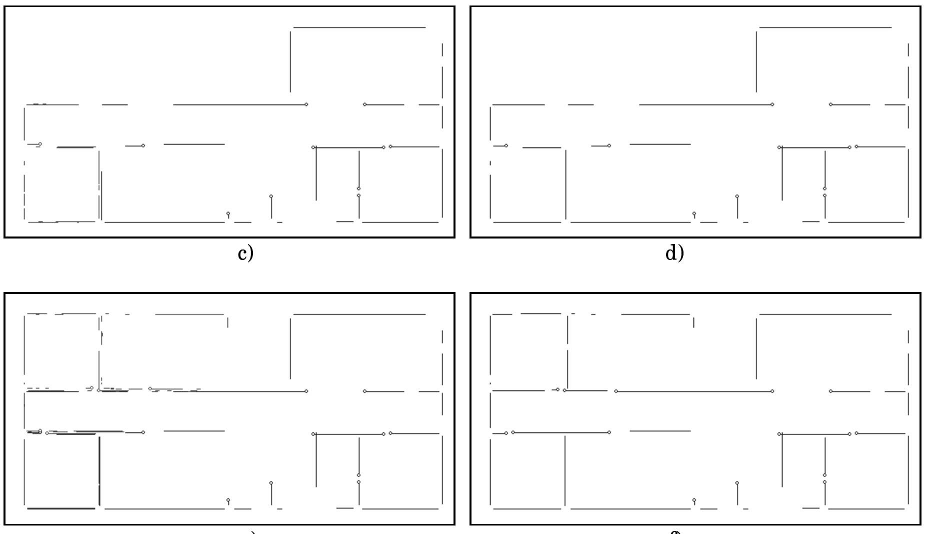


Figure 4.15: a) Communicated map from the exploration of the Env3 environment; b) Fused map obtained from the map in a); c) Fusion map in b) with a second communicated partial map; d) Fusion map from c); e) The fusion map in d) with the last communicated partial map; e) Fusion of the map in d).

For the previous example we have used a fusion threshold of 0.75 m —it is a parameter of the program. We have set this value smaller than the standard doorway size because, due to robots' error, some relative distances between doorway segments are smaller than 0.8 m. This decision is done online: the user in the host computer can evaluate the fusion process results and try different thresholds in order to obtain the most satisfactory map. Of course, this criterion is purely subjective, it simply depends on the common sense of the user but we use an example that clarifies this idea. Next Figure 4.16 shows two additional fused maps with different threshold values. They illustrate how the change in this parameter affects the resulting map. Map a) in the figure has been obtained by setting the threshold value to 0.8 m. With this value, a doorway near the right side of the map appears to be closed —or as a single wall— because its gap is smaller than 0.8 m. The same effect happens for most doorways in map b) because the threshold used to its fusion was 1.2 m. This value does not make much sense because it is bigger than the average size of a door, and therefore it is very likely that doorways are collapsed into single walls. And therefore, as we can see in the figure, most segments at both sides of doorways are fused. Since the user does not know the map of the environment, he or she can only make a hypothesis about what segments should fuse. Considering a threshold of 0.75 forces the right-most doorway to appear, and this is the reason for which the user decided that 0.75 is more suitable than 0.8 or 1.2. However, this only applies for this case. In general, the user has to find the compromise between not closing doorways and doing as much fusion as possible: the smaller value the user sets, the more conservative the map is and the lower fusion rate the map has.

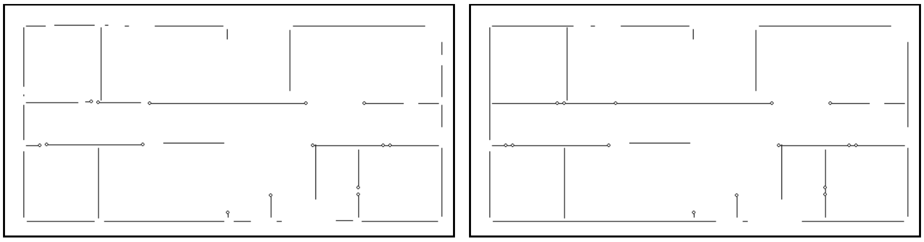


Figure 4.16: a) Fused map with threshold equal to 0.8m. b) Fused map with 1.2m as threshold.

4.2.2 Completion

As it has been previously described (see Section 1.1), the Map Generation algorithm calls the completion function with the current global map. This function is a post-processing function that tries to improve the global map as well as to identify some environmental features —i.e., singular points, corners and doorways. Completion applies heuristic rules in order to define these features from the relations among imprecise segments.

Heuristic rules are applied in the following complexity increasing order: first, L-corner rules; second, T-corner rules and, finally, doorway rules (see their specification in Sect. 1.1). We will use the fused global map from the Env3 environment in Figure 4.15 f) to see the results produced by the heuristic rules. The next Figure 4.17 shows this global map after the application of the L-corner rules.

By the changes in the singular points of the figure, we can see that 16 corners —out of 24 real corners— have been produced. L-corners have been generated by relating pairs of perpendicular segments. They can be distinguished in the image (Figure 4.17) because their singular points appear highlighted by black circles. The α threshold parameter value that is used in these results corresponds to 0.8 meters.

Besides light coloured singular points, which relate two segments and come from single applications of L-corner rules, dark singular points appear in the figure as the intersection of three different segments. Although they look like T-corners, they do not come from the application of T-corner rules but from applying L-corner rules twice. For the first L-corner rule application, one segment S_i is related to another segment S_j producing an L-corner $c_i=(p_i, S_i, S_j)$ with a singular point p_i. Afterwards, a second L-corner rule application results in a new corner $c_k=(p_k, S_k, S_l)$, which relates a pair of segments (S_k, S_l) that intersect at a point p_k. This new corner will be directly included into the corner list only if the p_k point is different to the previous p_i. Otherwise, an additional segment comparison is required in order to avoid repetitions of corners in the list. Two corners are considered to be different ($c_k \neq c_i$) when, at least, one of their segments is different (($S_i \neq S_k$ and $S_i \neq S_l$) or ($S_j \neq S_k$ and $S_j \neq S_l$)). Dark singular points in the figure correspond to two different L-corners that have one segment and a singular point in common.

We can still find another case of two L-corners sharing a segment in Figure 4.17. The difference in this case is that, due to detection errors, their singular points do not coincide. They are very close, though, and therefore they appear as a kind of double light point. They are not merged into a unique singular because L-corner rules do not handle segment alignment. (We can find such kind of double L-corner at the bottom of the figure).

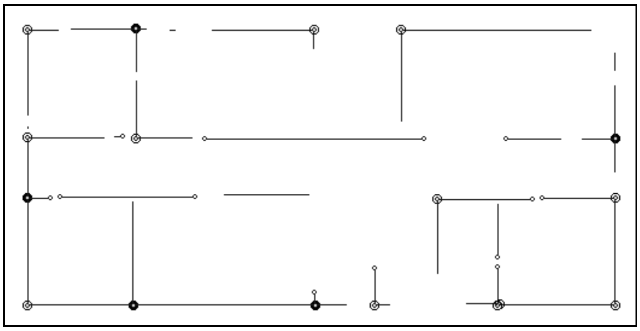


Figure 4.17: Completion of the L-Corners of the map in Figure 4.15 f).

T-corner rules are the second rules that are applied over the global map. In our example, they have only produced two T-corners between pairs of perpendicular segments —whose intersecting point is close to the extreme of one segment and far from both extremes of the other segment. These two T-corners appear as dark singular points in the following Figure 4.18.

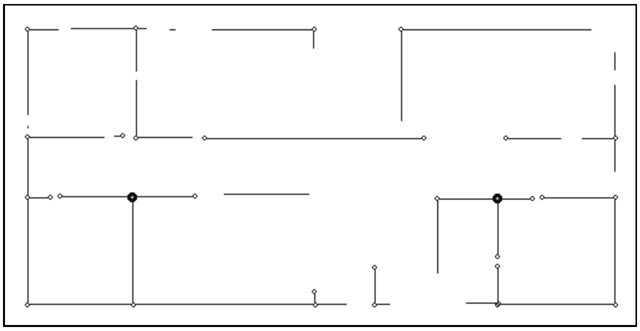


Figure 4.18: Completion of the map of Figure 4.17: application of T-corner rules with α = 0.8 m.

Finally, the third computation of features corresponds to the generation of doorways. Doorway rules (see Sect. 3.5.2) are applied assigning to α and β parameters a value of 0.8 meters. In the following Figure 4.19 they appear as two grey singular points linked by a grey arch. Five doorways out of eleven have been produced. For the remaining six doorways: one of them was not traversed by any robot—the one on the top of Env3—; four had a missing detected segment —that is, the robots only detected one of the two segments that define a doorway—; and the sixth doorway did not fulfil the requirement about the relative distance between segments (from the left side of the central horizontal corridor, it is the second doorway on the right).

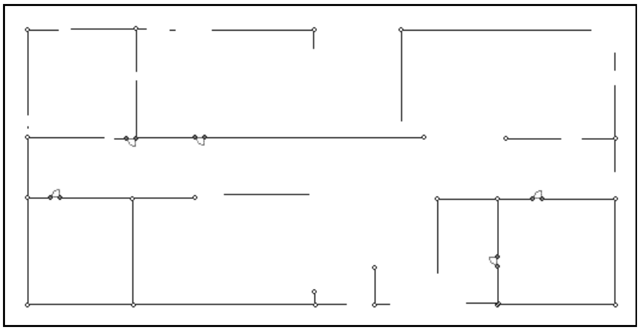


Figure 4.19: Completion of the map of Figure 4.18: application of doorway rules (α = β = 0.8 m.).

4.3 Random Exploration Strategy

The Robot Navigation section in the second chapter describes how robots show a partially random behaviour when exploring the environment. Randomness is present through probability values. On one hand, a turning probability defines when the robot must turn. And, on the other hand, a direction probability that defines a tendency to turn to one side more often than to the other.

Due to this turning probability, when the robots find a wall and follow it, they leave it according to its probability value —a minimum distance is forced, though— even if the end of this wall has not been reached. This strategy may seem to be counterintuitive, but it is good for the kind of coverage we are looking for. Considering that robots have very limited resources —as batteries, etc.—, we rather prefer them to find more than one wall than just finding one and following it until its end. Nevertheless, we are also concerned about the policy of "once you have found something try to extract as much information as possible". Therefore, we use different values in the probabilities to ensure that some robots will follow walls for "reasonable" distances. In this manner calm robots follow walls along longer distances than nervous robots. The reason to have nervous robots is that they tend to exhaustively cover an area, providing also information about free space in the environment.

Our exploration strategy of leaving a wall before its end tends to homogenise environment coverage. By homogeneous coverage we mean that detected environment features —together with their associated errors— are uniformly distributed on the environment. By default, our problem setting does not favour homogeneous coverage. This is due to that, since all robots start at the same initial position, it is more likely to happen that robots detect walls near this position than distant walls. In addition to that, the sooner a robot detects a wall, the smaller its associated error is. Taking this into account, our navigation strategy tries to balance this situation by leaving detected walls once they have been followed along a sufficient distance. This strategy increases the chances for far away walls to be detected with acceptable accumulated errors. Of course, it is reasonable to think that, if robots leave walls before their end, then, part of the information about nearby walls will be lost. This may look like losing close information in order to acquire some distant information. However, this strategy tends to obtain more information than it looses because, since close walls have high probabilities of being detected, a significant part of this lost information would have been resulted to be redundant when considering other robots' information.

These previous comments about random environment coverage are just intuitions. We have not exhaustively studied this problem because there are other works that already give mathematical soundness to random exploration and environment coverage. Some results and a brief description of some of these works are given in the following subsection.

4.3.1 Random Environment Coverage

In 1959, Nash-Williams presented the analogy between the resistance of electrical networks and random walks on graphs (Nash-Williams 59). Later (Doyle 84), this analogy was used for investigating the recurrence properties of random walks on 1, 2 and 3 dimensional grids. Two representative results of graph coverage by random walk give the following upper bounds to the coverage time of a graph (with m edges and n vertices):

- $O(m \cdot n)$ was given in (Aleliunas 79)
- $O(m \cdot \rho \cdot \log n)$ where the ρ is the resistance of the graph, assuming all edges to be 1-Ohm resistors (Chandra 89).

It is also known that this coverage time is significantly reduced if several random walkers are properly distributed in the graph (Broder 94).

Although these works about graphs are interesting because we can generate a graph from our maps (by discretizing them and defining a node for each resulting cell), we rather focus on the results of a recent work on continuous domains (Wagner 98). In the rest of this subsection we summarise some coverage results by Wagner et al. They concluded that:

- On the average, a random walk is not too bad compared to deterministic algorithms that use complex computations to calculate their steps.

Deterministic algorithms can be generalised as follows:

1. Discretize the environment in areas (defined by the range of the robot sensors) and go to the starting position.
2. From the current position, see if any neighbour area has not being covered:
   - if there is an area: go there, mark the area as covered and make a line between the previous and the current area.
   - if there is no neighbour area: backtrack using the lines.
   - if it is not possible to do backtracking: stop (end).

Random algorithms chose a random point around the current area.

Wagner et al. results are based on the definition of a relation between the cover time of a Markov process and the electrical resistance of the explored region. The cover time of a Markov process is defined as the average time that a robot needs in order to cover a given region area. It is computed considering that a robot senses (i.e., covers) a circular area around its position and that it visits a sequence of points such that the union of covered circles include the region. On the other hand, the electrical resistance of the region corresponds to the electrical resistance of its surface (e.g., the voltage between two points on the surface when applying 1 Ampere of current between these points). The relation between the cover time and the resistance of a region is established because of a previous work (Matthews 88), which relates the coverage time with the hitting time in a Markov process (were the hitting time is taken as the average time needed to move from a point to the circular neighbourhood of another point).

Considering an unknown area *R* that can be discretized in *n* unitary square cells we assume *n* to be the size of *R*. In general, we know that:

- In order to cover *R,* a robot that covers —i.e., senses— a circle of radius $\sqrt{2}$ must do a number of steps that is, at least:

$$\left\lceil \frac{3n}{4\pi + 3\sqrt{3}} \right\rceil$$

In addition, when the robot chooses a random direction at each step, they conclude that:

- The expected time of complete coverage $E[T^{PC}]$ of *R* is bounded by:

$$2n\rho \le \mathrm{E}\left[T^{PC}\right] \le 2n\rho \log n$$

Where ρ is the electrical resistance of *R*.

Wagner et al. also conclude upper bounds for the variance and standard deviation of the coverage time:

$$V[T^{PC}] \le 2^{11} n\rho \quad \text{and} \quad \sigma\left[T^{PC}\right] = \sqrt{V\left[T^{PC}\right]} \le 32\sqrt{2n\rho}$$

All these results stand for a completely random strategy that chooses a new robot direction for every step. In our case, this random direction is chosen after covering a random distance —which comes from the robot turning probability. Therefore, we could think in our exploration strategy as a biased random exploration such that, for each step, the probability of choosing the next direction as being equal to the previous direction is larger than the probability of choosing any other possible direction. In this manner, our exploration strategy fits the categorisation of random exploration, so that the previous results also apply for it.

4.4 The History of this Approach: some Pitfalls.

This first part of the thesis reflects the approach developed within a collaboration of the Artificial Intelligence Research Institute —where this Ph.D. thesis has been developed— with the Automatic Control department at the UPC (Technical University of Catalonia). The automatic Control department was building a group of autonomous robots while we were developing the map generation process. Considering that robots would explore highly structured environments with low density of objects, we chose a symbolic approximation to this environment-modelling problem.

In this manner, we first defined imprecise segments as a representation of the environment. Secondly, we specified the fusion algorithm in order to group the information —and therefore, simplify it. And we finally sketched the completion rules that generate some environmental features —corners and doorways. However, since we were developing these ideas in parallel to the robots construction, we had to use the previously described simulator to obtain data for testing our algorithms.

Once they had a stable version of one of the robots hardware, we were able to do the statistical error analysis shown at the end of the second chapter (Sect. 1.1). The testing of the robots navigation software was the next natural step. Unfortunately, a problem in the following of walls appeared during the testing in real environments. During the micro-servoing, it turned out that so many micro turns were done that robot developers decided to incorporate a compass in order to obtain the orientation of the robot —rather than to compute it by using odometry— because it accumulated unnecessary errors. Three robots had being developed by then, however, they never where completed because funding problems of this starting project at UPC did not allow to install the required compasses.

In this situation we decided to keep our research in the field of map generation but using our own simulated data. By then, we had advanced enough in our symbolic approach to be aware of some limitations regarding the completion process. These two circumstances yielded us to start a new approach to solve the same mapping problem. This new approach is the one presented in Part II. In the rest of this section we comment the limitations we found for a possible further work on symbolic map completion.

Further Work on Map Completion

Chapter 3 presents heuristic rules that define features of the environment by considering relative segment positions. However, from the results shown in Subsection 4.2.2 it is obvious that this completion process could be further developed. In fact, the completion concept itself has been reduced to just complete corners and doors, but it could be generalised to include other kinds of segment processing such as extending single segments or combining segments that were not grouped by the fusion process. However, even restricting the completion to generation of features, there are at least too more directions that need further development. On the one hand, corner and doorway rules could be refined in order to include more cases. And, on the other hand, other concepts could be generated as well. Next paragraphs comment some of these cases.

As Subsection 4.2.2 shows, there are T-corners in the environment that are defined in the map as two L-corners sharing a segment. Of course, it would be possible to have an extra rule that looked for these specific cases, erasing the corresponding two L-corners and generating a new T-corner. However, this has two problems. First, the generation of a T-corner only includes two segments —and therefore, it should be changed. And second, for those cases where the singular points of the L-corners do not coincide, the rule should first align some segments —we have previously said that alignment is not desirable due to its propagation effect. Furthermore, the T-corner generation needs more elaboration than just extending it in order to include more segments. This is because the union of two L-corners can lead to two different T-corners: if they share a segment, then it corresponds to a T-corner, otherwise, it is a cross-shaped T-corner. Next Figure 4.20 illustrates these cases.

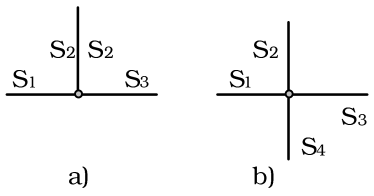


Figure 4.20: a) A T-corner that has been generated as two L-corners sharing the S_2 segment and their singular point; b) A cross-shaped T-corner that has been generated as two L-corners sharing their singular point.

There are cases where doorways whose walls have been properly detected are not generated because of the absence of a robot trajectory going through it. This is a reasonable condition because, if this was not required, the system would generate doorways that would correspond to real walls in the environment. However, this traversability information could be used by new rules to generate doors having a gap too big but with a robot trajectory going through.

From the combination of corners and doorways, the concept of a door-corner appears. An extra heuristic rule could generate them from situations where neither a corner nor a doorway can be generated. Two perpendicular segments cannot generate a corner if there is a robot trajectory passing through their closer extremes. Whilst a doorway cannot be defined for non-collinear segments. Next Figure 4.21 shows two examples of door-corners.

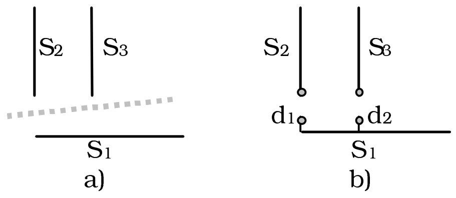


Figure 4.21: a) (S_1,S_2) and (S_1,S_3), perpendicular segments with a robot trajectory passing through their closer extremes. b) definition of two door-corners d_1 and d_2.

Other kinds of objects could also be defined by increasing the level of abstraction. For example, we could have the concept of room, as a union of walls having, at least, one doorway. Considering our structured environment, we would have more than 4 walls —or any other even number— and, we could also add the requirement of having the same number of corners as number of walls. The existence of a door guarantees that at least one robot has been inside the room. Therefore, as a complementary concept of room we could define the concept of obstacle as a closed union of walls. The obstacle definition would require that no robot had entered inside its area. This would mean that all walls had been detected from the outside of the obstacle.

The previous discussion illustrates that the number of possible cases can increase significantly, especially if we consider that we are just working with an orthogonal environment. Furthermore, we have not taken into account other factors such as the order of application of heuristic rules, segment errors or a more complex consideration of trajectory information. Thus, as we have already said, we decided to change into a less ad-hoc approach at this point of our research that will be described in Part II.

4.5 Conclusions

In this Part I we have presented an approach to the map generation problem. This approach is based on a troop of small autonomous robots that explore an indoor environment and a host computer that generates a plausible map of this environment.

The indoor environment is assumed to be orthogonal with a significant portion of passable areas —this is, the floor must be mainly flat. In addition, the environment must be basically static because our approximation generates a static representation of the environment. Nevertheless, objects with thin legs —i.e., chairs, tables, etc.— or persons can move because their sides will not be followed and therefore, they will not be represented. On the contrary, large obstacles —i.e., having sides long enough to be followed— are supposed to remain in their locations and are thus represented.

Robots explore the environment by following six different randomised behaviours. This strategy not only helps in distributing the accumulated error along the environment but it also performs a reasonable coverage of the environment. Furthermore, by assigning different values to robots' turning probabilities, allows to obtain different kinds of information such as wall positions or free space.

Robots start their exploration from the host position, they gather the environment information, and finally —when they have accumulated a pre-specified error—, they return to their initial position. Their returning trajectory is very conservative (it is basically the same than the exploratory one but without loops). But guarantees safe traversability and, for a further work, previously detected features could play the role of landmarks to compensate for accumulated errors.

All robots in the troop that, after exploration, return successfully, communicate to the host their partial maps. In fact, robots can also share their partial maps when two of them meet. We have not seen this situation in the results, though. None of our robots in the simulation was lost because the environments were completely passable, and therefore, shared partial maps were redundant. Nevertheless, without focusing on this co-operative task of map sharing, we still can say that all robots work in the same environment and with the same goal —obtaining information in order to generate a map— and, in that sense, they collaborate in the map generation task.

The host implements a symbolic approach to include into the global map of the environment all the information coming from robots' partial maps. This symbolic approach is based on a representation of detected information by means of imprecise segments. Segments are imprecise —

that is, we do not know their precise location due to robot errors— but not uncertain because they come from sure detections and subsequent followings. From the imprecise information that we have about the position of a segment, we define the corresponding fuzzy set that is represented by a membership function that gives, for any point in the segment error area, its membership degree to a real wall.

Together with the imprecise segments, other symbolic features such as singular points, corners or doors are represented. On this representation, the host applies first a process of fusion of segments and, afterwards, a map completion computation based on heuristic rules. Fusion is done considering the imprecise segment's errors and relative distances between the segment candidates for fusion. On the other hand, a completion process applies heuristic rules that present some problems when trying to cover all possible cases. However, this symbolic approach, still has remarkable advantages. Next subsection comments them.

Advantages of Symbolic Approximation

Considering highly structured environments with a low density of obstacles, a symbolic representation is especially desirable because the elements in the environment can be easily represented and the size of the environment does not limit its representation.

As we have already explained, the elements in the map are represented in a reduced memory space. Memory is allocated dynamically, so there is no need of knowing neither the number of elements that will be represented nor the number of relations they will have nor the size of the environment.

The representation of environmental features is based on dynamic lists containing elements that keep their relations by means of pointers. Pointers allow a direct access to the information as well as quick updating without a significant memory cost. In addition, these pointers are bidirectional and therefore, the access to information is very flexible. We can, for example, access all the segments with singular points in two different ways: We can follow the list of singular points and, for each singular point, access the segment/s it belongs to. Or, on the contrary, we can follow the segments list and check those segments having a non-empty list of singular points.

This symbolic approximation allows a direct categorisation of environment features into detected wall segments, singular points, corners and doorways. This distinction of elements benefits the use of all their implicit information. In this manner, it is direct to use all the singular points in order to detect potential doorways or corners because of their wall ending implication.

Imprecise segments are ordered applying the Quicksort algorithm with an increasing error criteria (it is known to be optimal). These imprecise segments constitute the key aspect of our symbolic representation. They do not only allow to represent imprecision through fuzzy sets, but also to define a fusion operation for pairs of segments.

Fusion is by itself an essential process that groups information and reduces imprecision. Fusion is a simple operation over pairs of imprecise segments that reinforces their common areas whenever they correspond to detections of the same real wall. Of course, the direct consequence of information grouping is the reduction in the number of segments and in the amount of memory used in the map representation. Nevertheless, what is more important, is the reduction it implies for the number of comparisons in the subsequent processing (the completion).

Completion constitutes the higher level process in the symbolic treatment of the map information. By using heuristic rules, it deduces different environmental features —more precisely, two different kinds of corners and doorways— that open a variety of future work possibilities. Just to depict some examples of future research, we list here several open directions without aiming to be exhaustive. First of all, generated environmental features can be used as natural landmarks for path planning. In addition, these same features can help in the symbol grounding process of a robot that moves in indoor environments. Moreover, since these symbols represent known human concepts, they can define a vocabulary for the communication of robots with human beings. And finally, they can help the robot to apply symbolic learning.

Part II: Using Possibility Grids for Structured Indoor Environments

Chapter 5

Simulation System of the Robot Exploration Troop

The first part of this thesis described a troop of autonomous robots and a host computer that collaborate in order to generate a map of an unknown environment. Robots explore the environment using a navigation strategy based on probability values. The results in this first part are maps generated at the host computer by using the exploration information from robot simulation.

For the second part of the thesis, a new simulator has been developed and a new approach to the map generation process is also presented. Basically, the simulation reproduces the hardware characteristics of the real robots. The main differences are that it implements a different navigation strategy and that it models the error obtained from the analysis of the real robots (described in Sect. 1.1).

Basically, both parts treat the same map generation problem —including the environment exploration problem. Nevertheless, they have been approached from a rather different point of view. A brief comparison can be sketched in two points:

- In the first part, robot exploration strategy is essentially random, whereas in the second part navigation is done by the co-ordination of several basic behaviours —it keeps a random component, though.
- The first part presents a symbolic (fuzzy segment based) approach to the map generation process, while this second part represents the environment by means of a discretization: a Possibility grid.

The current chapter describes the implementation of the simulator system and includes a detailed description of the former point. The second point, that is, the representation of environment maps, is the main issue of Chapter 6th.

Both parts have, therefore, important aspects in common. Nevertheless, the interest of different approaches increases when they exploit complementary features and if their results are useful for different purposes. Considering their differences, one or the other can be chosen on the basis of the nature of specific goals in the map generation process. We have already seen that our symbolic approach is helpful in extraction of environmental features as corners or doorways. On the contrary, a grid map representation is more appropriated for different map treatments. Chapter 7 concentrates in two subsequent processes: segment extension and path planning.

Finally, Chapter 8 returns to the simulation system in order to describe how the same behaviour-based navigation strategy can be used to follow the planned paths obtained in Chapter 7.

5.1 Introduction

Section 1.1 specifies the team and the task of the ANT project. The present chapter describes a simulator that follows the same specification, although some aspects in the general settings of the task have been redefined. Basically, this chapter is a technical description of the simulator. It describes the simulated environments as well as the simulated robots. Most of these descriptions are based on the ANT project, and all introduced variations will be punctually specified in the pertinent moment. These variations concern environmental assumptions as well as initial and final robot positions and, despite of their generality, they do not constitute significant changes in the problem statement. On the contrary, they can be seen as fairly independent from the two main approaches that are presented in this second part: Behaviour-based navigation and map representation by means of Possibility grids.

Since this chapter describes the simulator implementation, it is important to specify that this simulator is a PC application. It runs over the Microsoft Windows 3.1 and 95 Operative Systems, and it has been implemented in Borland 4.5 C++. The fact that it is developed in C++ implies that some of the subsequent comments will make some references to Object Oriented terms (Eckel 91) —as objects, their methods or inheritance. We cannot either avoid commenting some aspects about the user interface —as menus, dialog boxes, child or frame windows. In fact, our application presents a Multiple Document Interface —MDI (Heller 92)— developed using the classes from the Borland's Object Windows

Library —OWL (Borland 94). The explanation will be as much independent as possible from the classes actually used. Nevertheless, some of them are mentioned in order to illustrate and clarify the implementation.

5.2 Simulated Environments

5.2.1 Non-orthogonal Environments

The first part of this thesis assumes that office-like environments are orthogonal. For this second part we relax this restriction and, although we still assume that they are mostly orthogonal, we also consider and represent non-orthogonal features in the environment.

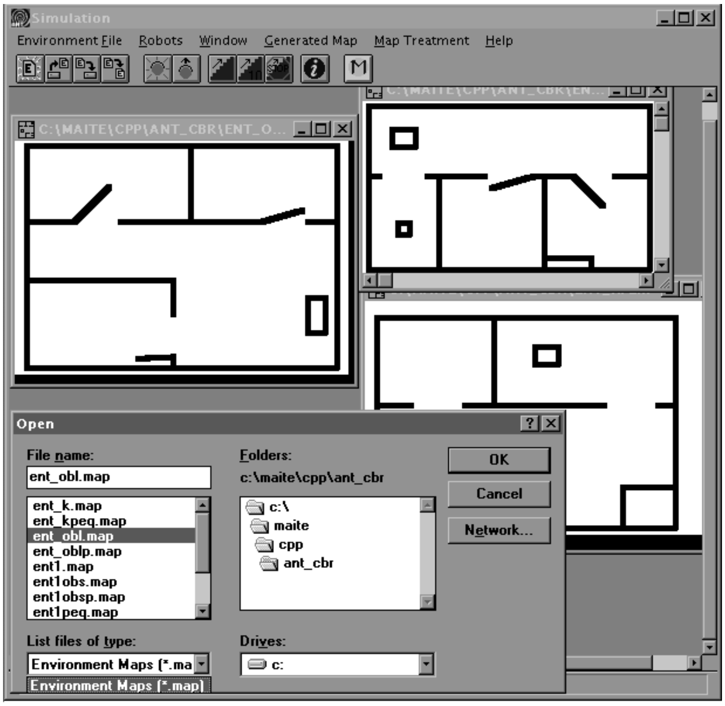
In the current chapter we will see that, although robot navigation follows a behaviour-based strategy, robots explore once again by moving randomly in free space and following doors —or obstacle edges— when detected. Therefore, our mapping process still considers only those environmental features having edges long enough to be followed by the robots (small obstacles, such as legs of chairs, desks, or people, are thus not represented). The fact of considering only relatively large environmental features makes the orthogonal assumption more reasonable. On the one hand, walls are usually connected by right angles, and on the other hand, pieces of furniture in an office —such as bookshelves or drawers— tend to have rectangular shapes.

Besides the obvious advantage of complexity reduction, the orthogonal assumption presents an extra benefit. In case a robot follows a wall —or an obstacle edge— without an exactly parallel trajectory, the resulting detected segment will not be orthogonal. In such a case, orthogonality allows the host to correct the given segment by generating its closer orthogonal segment.

Nevertheless, half opened doors appear so often in office-like environments that they forced us to relax the orthogonal restriction for this second part. Therefore, our mapping process considers now segments going in any direction. However, it still considers the orthogonal assumption for those detected segments that are almost vertical or horizontal, so that they are orthogonalized whereas oblique segments do not change orientation. This relaxation of the orthogonal assumption means a significant improvement to the mapping process because, as Yamauchi points out in his recent work (Yamauchi 98), it is still quite common to find in the literature systems that can only handle parallel or perpendicular walls.

5.2.2 Environment Simulation

Environments in the simulator are opened and saved through the user's Multiple Document Interface. Figure 5.1 shows a snapshot of the simulator with three opened environments together with the Open Dialog that will allow to open a forth environment. Environments are stored by means of *environment maps*, which consist of an array of walls or obstacle edges. As in the previous part, we do not distinguish walls from obstacle edges, they are assumed to be decomposable in rectilinear segments so that they are defined using the co-ordinates of their initial and final points. Furthermore, all the environments in the simulator are supposed to be flat, and therefore, these points are defined in a two-dimensional plane.



File name: ent_obl.map

ent_k.map
ent_kpeq.map
ent_obl.map
ent_oblp.map
ent1.map
ent1obs.map
ent1obsp.map
ent1peq.map

Folders: c:\maite\cpp\ant_cbr

c:\
maite
cpp
ant_cbr

List files of type: Environment Maps (*.map)

Drives: c:

Figure 5.1: Snapshot of the simulator when the user is opening several environments. Three of them are already open. Except for the one that is being opened, all of them have a granularity value of 2 cm. and considering them in clockwise direction, their sizes are 5.7m.× 4 m., 5m.× 3m., and 6m.× 4m. respectively.

The opening of an *environment map* implies the generation of an environment in the simulator. Although this statement might look straight forward, it is important to make the distinction between *environment maps* and *simulated environments*. An *environment map* is basically a file containing wall information whereas a *simulated environment* is the scenario of the simulation. Inside the client area of the main window of our simulation application, each *simulated environment* is created as a new child window (i.e., child of the main window) that inherits all the display properties of the MDIClient class. In this manner, we can describe a *simulated environment* as the screen of the simulation process. As in every Multiple Document Interface (MDI) application, there can be several environments simultaneously open, but simulation is only done in the *simulated environment* window that it is currently active.

5.2.3 Environment Class Members

Our simulator defines a number of object classes which is too large to describe all of them. Nevertheless, there are two classes that are fundamental for the system and we cannot avoid detailing them. These are the environment class and the robot class. Here, in this subsection, we enumerate the principal members of the former class:

- *Environment general features*. Their values come from the environment map (they are assigned by the user during its first definition). We consider a discretization of the real environment in a two dimensional grid, this grid is characterised by:
  - Its granularity, which gives the relation between the real and the simulated environments. Its default value is 2 cm meaning that each cell in the simulated environment represents two square centimetres of the real environment. All opened environments in Figure 5.1 have a two-centimetre granularity (a value of 1 cm is also very common, though).
  - Its x and y dimensions, together with the granularity, give the size of the real environment. Their default values are 5 metres for the x axis and 3 for the y axis. In the Figure 5.1, this is the actual value of the map on the right top corner. These values give the size of the white area, which includes all wall co-ordinates within the environment.

- Array *of walls*. This array is created from the segments in the environment map (which correspond to walls and obstacle edges). By default, a wall is considered to be 10 cm. wide. This value is set to be the same than the one in the previous part (see Sect. 4.1.1).

- *Grid map of the simulated environment*. As we have already mention, the environment is discretized into a grid. The size of the grid corresponds to the x and y dimensions of the environment divided by its granularity. That is, the grid is a matrix where the number of columns is the x dimension divided by the granularity and rounded to the upper natural (and respectively, the number of arrows comes from the y dimension). Each cell in the grid contain the following fields:

- Wall or obstacle presence
- Robot body
- Robot collision detector
- Robot Infra Red signal (IR)
- Robot presence signal
- Robot area of presence de tection

These fields take numeric values meaning the number of elements of the corresponding kind that exist for each particular cell. Having a coherent value assignment for these six fields is enough to simulate an environment with several robo ts moving inside.

Wall values are represented initially. When the user opens an environment map, the grid is defined and walls are marked inside the grid by assigning 1 to the wall presence field of the corresponding cells. Since both, wall and robo t body fields, represent physical occupancy, their values are incompatible and can only be 0 or 1. On the contrary, the presence of robot body or wall with infrared signal in the same cell can be used to model detections. For example, if the current position of a robo t is being represented and the fields corresponding to infrared signal coincide in the same cells than wall fields, this can be interpre ted as the robo t de tecting a wall.

Although each field is represented by just one byte, it is obvious that the size of the grid ne eds the allocation of a significant amount of memory. This constitutes the main reason for simulating environments with a limited size. As we have already said, our default size is 5m.$\times$ 3m., which corresponds to 879K.

• *Array of robots*. Robots are defined to belong to the currently active environment. Each environment can handle a maximum number of five robots. Next section 5.2 details how does the simulator represent and manage robots in the environment.

• *Display members and overridden inherited functions*. As we have already said, the environment class is derived from a Window class, and therefore it has inherited functions that help in the display of the window client area. Basically, the information to display is the representation of the grid map. This grid is updated after robots' movements. Therefore, In order to show the current state of the grid map, we have to synchronise robot's movements with the windows updates. This is done redefining some of the inherited display functions and creating some auxiliary elements.

Initially, the function SetupWindow() is redefined so that it creates one Memory Device Context —a memory area where it is possible to draw the information that will be displayed in the client area of the window—, one Bitmap and all the Pens and Brushes necessary to draw every displayed feature. This function also marks the walls in the environment map in both, the grid map and the Memory Device Context.

Afterwards, each time a robot is added or the robots move, the representation of the robots must be changed. And, once again, this is reflected in the grid as well as the screen. In the next Section 5.2 we will give details about how robots update the grid. Regarding the screen updating, the function UpdateWindow() sends a WM_PAINT message to the window EventHandler, which calls the inherited function Paint with the window's Device Context as parameter. This Paint function is redefined so that it updates the representation of the robots in the Memory Device Context (using the Bitmap, Pens and Brushes) and transfers it into the Device Context of the window (this is what actually updates the screen). The translation from each cell in the grid into each pixel in the Memory Device Context is done by assigning priorities to the cell labels. That is, each pixel takes the colour from the label with the highest priority in the corresponding cell. (We will further explain this in the following section).

Finally, in order to add some more window capabilities, several functions have been created —or overridden from the Window class. Some of them manage the window with flags (this is the case of CanClose(), which uses the IsDirty and IsNewFile flags); while others use messages (for example, EvSize and AdjustScrollers() pass messages to LayoutWindow).

• *Interface functions*. The user guides the execution of the simulator by activating all the options that can be selected for an environment. These options are supported by a number of functions and they are available to the user from the menu in the main window frame. They can as well be selected by mouse —opening the menu or the equivalent tool bar buttons— or by pressing the quick access control keys. The Event Handler uses several Response Tables that are defined in order to establish the connection between the messages generated by the interface elements —when chosen by the user— and the functions that actually perform these options.

Up to this point, we have already seen that an environment class has several components that are related to different capabilities. Seeing an environment as a container of the environment map, it can be created, saved, saved with a different name, and loaded. All the options that need some user information open different dialog boxes and use predefined information to validate the inputs. In this manner for example, the environment maps are saved in files with the '*.map' extension.

On the other hand, environments are windows that display its grid contents. From the windows point of view, the user can:

- move, select, expand, resize, minimise, or close an environment window with the mouse as a regular window
- automatically arrange icons of minimised windows,
- close all windows in the application,
- arrange windows in different positions: title or cascade, and
- select and bring into the first plane of view a specific window name

Finally, an environment has up to 5 associated robots. There are two menu options that are related to robots: Add Robot and Start Robots. A robot cannot be added while the simulation of other robots is being executed. Therefore, the user is expected to, first, add robots and then, start their exploration. A robot can be started just once, therefore, if after a simulation the user adds new robots, only these are started, while the rest of robots remain stopped in the environment (they act as new obstacles).

• Constructor and destructor member functions. Although we are describing the environment class as if their instances were huge objects, it is not the case. Most of the elements that have been described are just pointers to other objects that come from different classes. In this way, for example, the environment does not have an array of robots: there is one array of robots class, and the environment just has a pointer to an instantiation of this class. Walls in the environment map, memory device contexts or the grid map are some more examples of pointed objects. Considering that some of them make a significant allocation of memory, and since most classes must be instantiated when creating the environment object, the creator and destructor are very important functions that are in charge of all related classes management.

5.3 Simulated Robots

As we have already said in the Introduction of this chapter, the simulator described here is based on the same problem settings that those in the ANT Project. However, there are two slightly different assumptions about the initial and final positions of the robots in the environment.

- First, we assume that robots can start their exploration from different positions. From a simulation point of view, this simply means that robots are initialised in positions defined by the user. This is done in order to increase the environment coverage. (We already described the result by (Broder 94) in the 4.3.1 Random Environment Coverage section).

- Second, we assume that robots finish their exploration when they have accumulated more error than a given threshold. This threshold is set in the system meaning that when a given information has an associated error bigger than this value, we consider this information to be useless. In this way, the robots do not keep exploring the environment during its way back to their initial position because they would obtain useless information about features that have been already detected. In fact, this simulation does not implement the way back. On the one hand, it could have been done in the same way than in the first part, by following the exploration path in an inverse order and suppressing trajectory loops. But, on the other hand, since robots start in different positions, to go back to a position different from the host position does not seem really necessary.

Both assumptions, initial and final robot positions, would require in real world —not simulated— a reliable communication system between the host computer and the robots. This system should allow the communication between a robot and the host by other means than infrared signals (that is for example, radio). In this way, a robot could receive its initial position from the host (this position would be set by the user) and could send its partial map to the host once it had explored the environment.

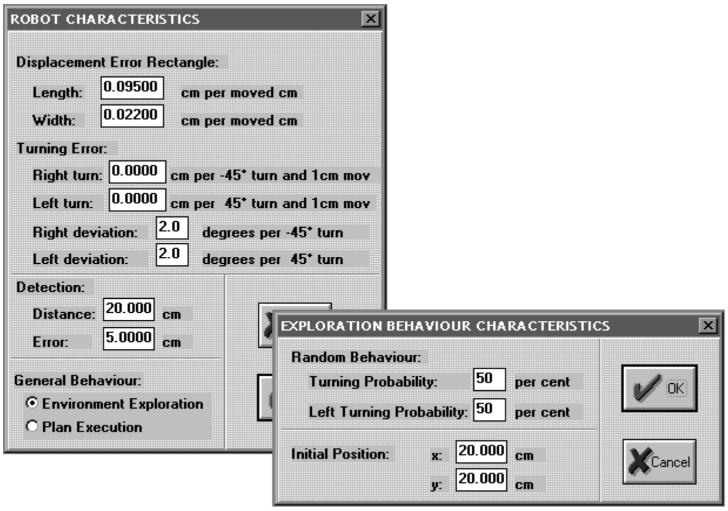
5.3.1 Defining New Robots in the Environment

Robot Characteristics

When defining a new robot that will explore the active environment, the simulator asks the user to input the following robot characteristics —which appear listed subsequently (see Figure 5.2 below). Their default values correspond to the ones obtained from the real robots (see the Error Analysis in Sect. 1.1):

- The displacement error that is perpendicular to robot's trajectory. Displacement error is modelled by a rectangle that increases proportionally with the covered distance; this error refers to its length. Its units are in centimetres. In this manner, its default value corresponds to 0.095 cm. meaning that, for each covered centimetre, the simulated robot accumulates an error of 0.095 cm. (this value comes from the 28.58 cm error that the real robot accumulates in a 3 m. displacement).

• The displacement error that is parallel to robot's trajectory. In terms of the rectangle error, this error represents its width. Again, its default value comes from the real robot error analysis in chapter 2 and is equal to 6.59/300=0.022 cm for each cm of robot displacement.


ROBOT CHARACTERISTICS

Displacement Error Rectangle:

Length: 0.09500 cm per moved cm

Width: 0.02200 cm per moved cm

Turning Error:

Right turn: 0.0000 cm per -45° turn and 1cm mov

Left turn: 0.0000 cm per 45° turn and 1cm mov

Right deviation: 2.0 degrees per -45° turn

Left deviation: 2.0 degrees per 45° turn

Detection:

Distance: 20.000 cm

Error: 5.0000 cm

General Behaviour:

[✓ Environment Exploration] Plan Execution

EXPLORATION BEHAVIOUR CHARACTERISTICS

Random Behaviour:

Turning Probability: 50 per cent

Left Turning Probability: 50 per cent

Initial Position: x: 20.000 cm

y: 20.000 cm

Figure 5.2: Dialog boxes for the definition of a new robot in the active environment. On the left, general characteristics. On the right, the dialog that appears if the robot is chosen to explore the environment.

• Right turning error. It is considered to be a segment approximation to the angular error in the direction perpendicular to the displacement. It is measured in centimetres, in terms of length of the segment that appears as a result of a –45 degrees right turn and a robot displacement of 1 cm. Since in our statistical analysis this error turned out to be a constant deviation, the default error has been set to 0°. However, as we will see further in this section, the simulator is able to include this error in the size of the displacement error rectangle.

• Left turning error. It is equivalent to the right turning error. Its default value is, as well, zero degrees.

• Right turning deviation. It reflects a constant deviation for every right turn. As we have explained, the error analysis resulted in a 2-

degree deviation on the left (positive) for each –45° turn (right), and it increases proportionally with the turning angle.

• Left turning deviation. This orientation deviation is equivalent to the right turning deviation. Furthermore, its default value is exactly the same: 2-degree deviation on the left (positive) for each 45° turn. (From these results we can assume that the studied robot has a minor built-in problem in its wheels that turns out in a constant left deviation).

• Infrared detection distance. Its value corresponds to the infrared range. Its units are also centimetres and its default value is 20 cm.

• Infrared detection error. It is an estimation of the Infrared error. Although we did not show any statistical information about this error, it was experimentally estimated to be 5 cm for white surfaces and regular light conditions. This is also the default value in the simulator.

• Turning probability. Following the ANT Project specification in the first part of this thesis (Section 1.1), robots explore the environment using a random strategy. Robot trajectories consist of turns between fixed orientation displacements. Random decisions about the frequency of robot turns are based on this turning probability. Its values determine the length of the robot displacements. These values are in the range of 1 to 100, being 50 the default value (Section 6.1 gives further details).

• Left/Right turning probability. Once the robot navigation strategy has decided to turn, this left/right turning probability chooses the new robot orientation —this is done if the environment does not force any particular turn. Again, the probability value must belong to the [1,100] interval and by default it is assigned to 50 meaning that robot will choose to turn right as often as to turn left. A probability grater than 50 implies that the robot will turn left more often, and conversely, a probability smaller than 50 will force the robot to turn right with a higher frequency (this is approximate, Section 6.1 details the computation).

• Initial position. It is specified in grid co-ordinates —being the (0,0) at the left lower corner —, and corresponds to the position of the robot's body geometrical centre. This position must be within a free area of the environment. The simulator does not accept any initial position causing a part of the robot to be placed in an occupied place or outside the limits of the environment grid. (Occupancy can be due to walls, obstacles or other robots).

Although the system does not ask them, there are some other robot features that play a role in the representation of robots. They appear hidden to the user because most of their assignments require some knowledge about the system, and our aim is to keep the interface as simple as possible. They are the following:

- Velocity. Although this characteristic usually refers to the robot velocity in real environments, in the case of the simulator it refers to the number of steps that a robot performs before the screen is updated. Its value is both related to robot speed and the speed of the computer running the simulator. In our application, this value is set to 1 so that the computer shows each step in the robot navigation, but if screen updating is slow, it can be set to 2 (or any other positive integer) and the screen will be updated for every two steps. This effect causes the user to think that the robot moves at a higher speed.

- Initial orientation. Each robot has a vector defining its initial orientation. It is defined in robot co-ordinates and it points to (0,1). That is, the robot is parallel to the y-axis of the environment.

- Shape. In fact, the simulator has been developed to represent circular robots only —which is a close approximation to the real robot shape. However, it would not be hard to change the code in order to represent other simple shapes as polygons. Further in this section, we will see how the robot sensors are drawn whenever the robot changes its orientation. Of course, a circle is invariant with respect to orientation so that the simulator does not need to redraw it. If it had another shape it would be simply treated in the same way than the robot sensors.

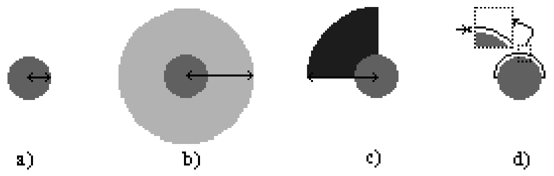


Figure 5.3: Black arrows define default robot characteristics: a) size; b) presence signal range; c) range of signal detection; d) distance between the robot body and the collision detection band.

- Size. This is a value that depends on the shape of the robot. In our case, robots are circular, and therefore, its size is specified by means of its radius (see Figure 5.3 a)), which is set to 10 cm. However, this size is automatically scaled to the granularity of the environment. In this manner, when an environment has been defined with granularity 2, robot bodies are shown as circles having radius of 5 pixels. (Obviously, all robot characteristics are automatically scaled as well).

• Presence signal broadcast. It specifies the range of the signal that each robot emits around itself so that it allows other robots to detect it (see Figure 5.3 b)). As in the ANT project real robots, it has a scope 360° of, and is represented by a circle with a radius of 30 cm.

• Presence signal detection. Each robot can detect other robots' presence signal within a scope of 90 degrees starting at its front and going towards its left. This arch of circle has been defined with a radius of 31 cm (see Figure 5.3 c)).

• Collision detection. The sensor for collision detection in the real robots is a flexible band anchored by two switches to the sides of the robot. In this manner, when an obstacle bends the band, it clicks one or both switches and the collision is detected. This band is simulated by half circumference that is 2 cm away from the robot's body (see Figure 5.3 d)).

Representing Robots in the Environment

When the user chooses the option of adding a new robot to the active environment, the system uses the previous characteristics —conveniently scaled— to define a new robot. Since each environment has a pointer to an array of robots, a new robot implies the construction of a new robot object and its subsequent addition to the array of robots —pointed by the environment. Nevertheless, the creation of a new robot also involves its subsequent representation. The robot has several elements that contain a representation of its body, sensors, and signals in the appropriate scale. When initialising the robot, the simulator computes the size of the area that a robot representation requires. Afterwards, it generates robot representation elements with the same area dimensions: Bitmap, Memory Device Context, and matrix. The robot Bitmap and the Memory Device Context are used to represent the robot in pixels; and the matrix represents the robot in the same terms than the grid map cells of the environment.

Since each pixel in those memory areas can take a unique colour value, the colour assignment is done in the same order than it would be done for drawing the robot on the screen. First, the pixels laying inside the presence signal circle are coloured in light blue (or light grey if B/W). Second, the system draws in dark blue (or dark grey) the detection signal, whose shape is a 90° arch. Thereafter, the robot body is represented by a medium grey circle and the collision band is painted as a black semi-circumference. Finally, the infrared sensors are 5 red radial lines going out of the body. Figure 5.4 shows the resulting robot representation. As we can see, the drawing order is equivalent to a priority colour assignment. In this sense, the later a pixel is drawn, the higher is its priority.

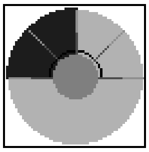


Figure 5.4: Simulated robot representation

The assignment of the labels to the grid is done similarly to the colour assignment. The system uses equivalent drawing algorithms for choosing the cell that will be labelled. However, the order here is not important because each element generates a label assignment to a different field of the grid cells. That is, when representing a robot element, the label assignment does not overwrite any label referring to any representation of a different robot element. The structure of the cells in the grid is very similar to the structure of the cells in the environment grid map (see Sect. 5.2.3). The only difference is that robot grid cells do not contain a field for the wall representation. In this manner, when marking the body, the detection scope, or the collision sensor, a '1' label is assigned to the respective cell fields. On the contrary, the presence signal can have two different labels, the '1' label is used to broadcast the robot's presence and a different label —'5'— indicates that the robot has already found another robot and is waiting to establish the communication. The label assignment to the infrared field (IR) in a grid cell is slightly different. There are 10 different labels: we need five to distinguish the sensor position (90° left IR, 45° left IR, front IR, 45° right IR, and 90° right IR); and in addition, for each sensor, two labels are used to differentiate the signal range that is 'near' the robot from the rest of the signal (which is 'far' from it). We will see in the next subsection how these labels are used to simulate robot sensing.

As an example of value assignment, Figure 5.5 marks three different positions in the robot representation. For the position number 1, the memory area has associated a grey colour for its corresponding pixel, whereas the robot grid cell has three different fields with label '1': robot body, presence signal and detection scope. Considering the position number 2, the memory area contains a light red pixel, and the robot grid fields without null values are: presence signal and detection scope have label '1', and the infrared field has the label 'far front IR'. Finally, position 3 corresponds to a dark red pixel in the robot memory area whilst the analogous cell in the robot grid has the following non-empty fields: collision sensor and presence signal have label '1', whereas the infrared field takes the label 'near 45° right IR'.

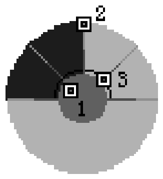


Figure 5.5: Simulated robot representation

Both, memory area and robot grid, are used to update the environment representation (memory areas update the environment window whilst robot grid updates the grid of the environment). Therefore, the system needs to compute the position where they will be added. This position is relative to the environment and is computed by subtracting from the robot position half the size of the area dimensions. The inclusion of the robot memory area (Memory Device Context) into the environment Memory Device Context is done by a bit-block transfer (which is a function that copies colour bits from a source rectangle to a destination rectangle) using the Boolean AND combining operation. Concerning the inclusion of the robot grid inside the environment map grid, we have to take into account that, although their field names that describe the robot coincide, these fields take different values. In the previous Section Part I1.1, we saw that environment cell fields take numeric values meaning the number of elements of the corresponding kind that exist for each particular cell. Therefore, all labels in a robot cell increase in 1 the field values of the corresponding environment cell. As the previous section Part I1.1 mentions as well, some of the values of the fields are incompatible due to physical occupancy. In this manner, the addition of a robot fails when it tries to increase wall or robot field values for environment cells that have previous non-zero values in any of these fields.

Next Figure 5.6 shows an environment with three robots that where added in different positions. From the bottom left to the top right robot, their positions in local co-ordinates —going from (0,0) to (285,200)— are respectively: (40,20), (80,100), and (262,157). The figure shows the environment represented by pixels, the colours for each pixel is chosen depending on the existing labels that its corresponding cell in the environment grid has. We have marked two pixels on the top-right robot in order to illustrate the label assignment. The pixel on the right is black because it corresponds to a wall. The cell in the environment grid that corresponds to this black pixel has the following values:

- Wall or obstacle presence = 1
- Robot body = 0
- Robot collision detector = 0
- Robot Infra Red signal (IR) = 1
- Robot presence signal = 1

- Robot area of presence detection = 0

On the contrary, the pixel at the front of the robot is red and represents the frontal infrared sensor. Its equivalent cell has set to 1 the fields describing the collision detector, the infrared signal, the presence signal and the presence detection area (wall presence and robot body have 0 value).

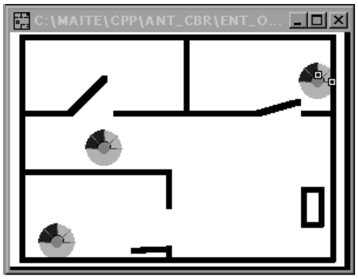


Figure 5.6: Initial position of three robots that have been added in the same environment. Robot that appears when adding the robot specified by the dialog box defined in the previous figure.

5.3.2 Moving Robots in an Environment

Once robots are represented in the environment, the user can decide to trigger their movement. Robots' movements are driven by actions. And actions are chosen so that they fulfil some goals taking into account the environment perception. The strategy to define actions is explained in the next chapter. What we see here, in this subsection, is how do we specify actions and how do we simulate their execution. But previously, since sensing is an essential aspect of action determination, we describe the robot sensing simulation.

Sensing Simulation

In order to know the sensor readings of a robot in the environment, the simulator compares the robot's grid together with the environment grid map. Basically, it consists in a value comparison between cells in the robot's grid and those cells in the environment grid map representing the same area. Sensor outputs are stored in a structure called Sensors (in fact, it is an object specified in an Object Oriented language). This structure, contains seven fields that take character values and correspond to:

• Sensors.collision: its value can be 1 or 0 depending on the existence or not of collision with a wall or another robot.

• Sensors.presence_detection: by default its value is 0, and if the robot detects another robot, it takes value 1 if the detected signal means presence or value 2 if the signal indicates that the other robot is waiting for communication.

• Left infrared (Sensors.leftIR): as the rest of infrared sensors, a 0 value represents the absence of physical detection —of wall, obstacle, or another robot—; a 1 value appears when there is a 'near' detection —that is, something has been detected to be close to the robot— and finally, 2 describes a 'far' detection (when something is detected at a distance that is bigger than half the infrared range).

• Oblique left infrared: it corresponds to the Sensors.45leftIR field and, as the left sensor, it can take values from the {0,1,2} set.

• Front infrared: it is represented by Sensors.frontIR. Once again, its values belong to the [0-2] interval.

• Oblique right infrared. It is equivalent to the 45° left infrared: Sensors.45rightIR$\in$ {0,1,2}.

• Right infrared: It is equivalent to the left infrared sensor field, its name is Sensors.rightIR and it takes values among 0,1 or 2.

The algorithm that assigns the values to the fields of Sensors compares all cells c_r in the robot grid with their corresponding cells c_e in the environment grid map. Basically, this is done by checking whether values at the c_e fields come from the labels of c_r cells (that is, if they come from the same robot grid) or, on the contrary, there is something else in these environment cells c_e. If this is the case, it is possible to conclude that the robot is detecting something. The comparison is done between specific fields in order to assign values to Sensors. In this manner, when the algorithm considers a robot cell having a particular sensor (that is, the corresponding field is not 0), it only checks if the corresponding cell in the environment has the type of presence that this sensor can detect. The robot can detect three different events: collision, detection of the presence of another robot, and infrared detection. Collision is detected when a robot cell that contains a collision detector has a corresponding environment cell with a wall, another robot, or another collision detector —i.e., robots can collide by their collision detectors. In the same way, in order to detect the presence of another robot, robot cells having the presence detection must coincide in the environment with the emission cells of another robot. If it is the case that the emission signal coming from the other robot means that this other robot is waiting for communication, then the value of Sensor.presence_detection is set to 2. Otherwise, it is set to 1.

Concerning infrared (IR) detection, it is richer in distinctions than the previous detections. Since there are 10 different labels for distinguishing IR sensors in a robot, when comparing a robot cell with the environment, it is possible to differentiate among 10 different IR robot detections that are represented in the last 5 fields of the Sensor structure. For example, if the robot cell has the IR field with a value of 'far front IR' and it coincides with an environment cell with wall field value set to 1, thus, the sensor field Sensor.frontIR takes a 2 value.

Finally, a brief comment about Sensor field values assignment. Although comparisons are made cell to cell, the Sensor structure contains values representing the whole robot grid, and therefore, some order and priorities must be taken into account when assigning Sensor's field values. The sensor reading algorithm can be depicted in the following way:

```
Sensor_reading()
{   For every cell $c_r$ in the robot grid repeat:
    {   Find its corresponding cell $c_e$ in the environment grid map,
        robot = $c_e$.robot_body– $c_r$.robot_body
        if($c_r$.collision_detector > 0)
        {  collision = $c_e$.collision_detector – $c_r$.collision_detector;
           if($c_e$.wall>0 OR robot>0 or collision>0)
             Sensor.collision = 1;
        }
        emission = $c_e$.emission – $c_r$.emission
        if( $c_r$.detection > 0 AND emission > 0)
        { if (emission $\geq$ 5) emission = 2  //2 means waiting for communication
          else emission = 1;          //1 means presence
          if(Sensor.presence_detection<emission)  //2 has more priority than 1
             Sensor.presence_detection = emission;
        }
        if($c_r$.IR>0 AND ($c_e$.wall>0 OR robot>0))
        {  ir=0;
           check the label of $c_r$.IR
          { case 90° left IR: if it is 'near' then Sensor.leftIR = 1
                              else if (it is 'far' and Sensor.leftIR $\neq$ 1)
                                   then Sensor.leftIR = 2
                              end case.
            case 45° left IR: if it is 'near' then Sensor.45leftIR = 1
                              else if (it is 'far' and Sensor.45leftIR $\neq$ 1)
                                   then Sensor.45leftIR = 2
                              end case
            case front IR: if it is 'near' then Sensor.frontIR = 1
                           else if (it is 'far' and Sensor.frontIR ≠ 1)
                                then Sensor.frontIR = 2
                           end case
            case 45° right IR: if it is 'near' then Sensor.45rightIR=1
                               else if (it is 'far' and Sensor.45rightIR ≠ 1)
                                    then Sensor.45rightIR = 2
                               end case
            case 90° right IR: if it is 'near' then Sensor.rightIR = 1
                               else if (it is 'far' and Sensor.rightIR ≠ 1)
                                    then Sensor.rightIR = 2
                               end case
          }
        }
        If (all fields in Sensor==0) then return 0
        else return 1
}
```

Robot Actions

Actions are the objects that drive robots' movements. Basically, our actions define the next position where the robot should move. In this sense, they can be seen as robots' subgoals. This subgoal position is defined in terms that are relative to the robot. They are the orientation that the robot must take in order to face the subgoal position and the distance that it must move once it is facing this direction.

However, an action has other members that help in the supervision of its execution as well as in the definition of subsequent actions. An Action has the following structure:

- Distance. Euclidean distance that the robot must cover in order to consider that the Action has been successfully executed (in this sense, distance coverage acts as a completion condition).

- Displacement. Intermediate counter that keeps track of the distance that the robot has actually covered from the action definition until the current simulation step.

- Orientation. Direction that the robot must face before performing the displacement. This orientation is specified in degrees and has the origin in the initial orientation of the robot. (Naturally, we can specify turning actions without implying robot displacement. This is done by assigning a 0 value to the distance).

• Sensors. This is the Sensor structure defined above. It stores the sensor readings that the robot has while executing the current action.

• Current behaviour and previous behaviour. The next chapter will describe the Behaviour-Based Navigation Architecture that the robots use to explore. There, we will see that basic behaviours define the actions and supervise their execution. These fields keep information about the behaviours involved in the current and previous actions.

• Velocity. It corresponds to the robot velocity. As we have already mentioned, it refers to the number of steps that a robot performs before the screen is updated.

• State. In the supervision of the action execution, state is a flag that indicates the action performance. It can take 6 different values:

- In case a robot is definitely stopped, the state value becomes –1.
- Initially, when no action has been defined yet for a robot, state takes an special default value (which is 0 in our case).
- Value 1 stands for a regular progress of the action execution.
- In some action execution cases, the action is still valid, but it needs a slight redefinition —as for example, a slight change in the direction preserving the distance. State = 2 indicates such situations.
- After an action execution that has been successfully concluded, state is set to 3.
- If during its execution, the action is interrupted, the state gets the value 4.

Action Execution

Up to this point, we have seen that actions are basically a distance to cover in a certain direction. Actions are executed in a number of steps that depend on the distance to cover. For each step, the robot moves forward a unitary displacement that is a constant of the system —in our case, it has been set to 2 cm. For the first step of an action execution, the robot turns and makes its first displacement, and for the remaining steps, it just has to advance in the same direction until the total distance has been covered. This simplifies the representation of the robot, because the robot needs to change its local representation —its memory device context and its grid— for the first step only. The remaining steps just require a change in the position of the robot —not the orientation—, and thus, a change in the origin co-ordinates of these local representations (i.e., the position that relates the local robot representation with the environment).

Each one of these steps consists of a computation of the new position and an updating of the robots' representation. Moreover, a new robot position also implies the treatment of possible inconsistencies and consideration of new sensor information. Regarding inconsistencies, they appear if a robot is placed over a physically occupied cell. If this happens to be the case, then the simulator interprets it as an irreversible failure and the robot is definitively stopped. This is reflected by assigning a –1 value to the action state. For the navigation strategy that we have developed we have not found this kind of contingencies. However, a similar situation appears when using open environments and the robot moves beyond the limits of the environment. (By open environments we mean environments without walls surrounding their limits).

Next algorithm describes the Action_Execution function. It first updates the robot representation. And based on this representation, it can update the sensor information and the state of the action, which will be completed if the distance has been covered. (Notice that, since turns are done without changing position and since robots have circular shape, there will be never any contingency for turning actions).

```
Action Execution()
{ turn = Action.orientation– Robot.orientation
  i=0
  Repeat while(i<Action.velocity AND Action.state=1)
  {  i=i+1
    if(Action.distance > 0)
    { compute the new position of the robot after moving an unitary displacement
      compute the new origin for the local representations of the robot
    }
    remove the robot representations from the environment.
    if (turn >0 AND i=0)
    then redraw the new orientation of the robot in its local representations
    transfer the robot local representations into the environment
    if (no physical collision was detected during the transfer)
    {  read the robot sensors
       if (Action.distance=0) then Action.state=3
       else
       {  Action.displacement = Action.displacement + unitary displacement
          if (Action.displacement ≥ Action.distance) then Action.state=3
       }
       check if the sensors gave a reading that forces any change in the Action.state
    }
    else block the robot (Action.state=–1) and recover the previous environmental and local
  representations
    Update Action.Sensors with the obtained sensor readings
  }
}
```

The next Figure 5.7 illustrates the execution of consecutive actions. Initially, the robot has been created with the default characteristics, probability values equal to 50 and initial position —in local co-ordinates— of (200, 40). This figure shows a sequence of snapshots of the simulator screen. With the aim of focusing only on the details in which we are interested, we have cut the snapshots so that they appear as a sequence of intermediate robot positions in the robot's trajectory. The first image (Figure 5.7 a)) shows the initial position of the robot inside the environment it belongs to. Remaining images show different steps during the execution of consecutive actions. None of these steps involve sensor readings, hence, actions have been generated randomly (without considering sensor readings). We give more details about action computation in the next chapter. By now, we just specify the distance and orientation of the actions in the figure. The first action has a distance value of 36.8 local units and an orientation of 44.9 degrees. Local units are defined by the environment granularity —which in our case corresponds to 2 cm.—, and, since unitary displacements are set to 2 cm, the robot moves 1 unit per step. Figure 5.7 b) shows the execution of the first step of this first action. During the execution of this first action, the robot turns in order to face the action orientation and moves afterwards one unit in this direction. A turning implies a redefinition in the robot representation, whereas a displacement implies a change of its origin. Therefore, the robot is drawn facing the 44.9° orientation and the representation is added into the environment starting at position (183, 25). The representation is a squared area of 33 units, and thus, this origin is defined so that the centre or the robot body is at (199, 41). These co-ordinates correspond to the rounded values of the robot's current position (199.3, 40.7). For the following 35 steps, the robot moves in the same direction, increasing by one unit the displacement field of the action. Figure 5.7 c) shows the robot during the execution of the last step of the first action. At this moment, the displacement field takes the value 37, and since it is bigger than the distance field, the state value of the action is set to 3 meaning that the action has been successfully completed. The second action that the robot must execute has assigned a distance of 33.3 units and –49.8°. Again, the first step to execute is a turning and a unitary displacement. Figure 5.7 d) shows this situation. This figure also depicts the last executed action by means of a black line. This line is automatically generated by the simulator and acts as a trace of the robot trajectory. Each line corresponds to an action, and it is not drawn until the action is completed (it can be successfully ended or just interrupted). Figure 5.7 e) shows the last step of the second action. Once again, the displacement is bigger than the distance and the action ends. And finally, we comment a third action. It has 46.9 units of distance and –96.6° of orientation. In Figure 5.7 f) we can see the new orientation of the robot and the track of the two previous actions. Last image (Figure 5.7 g)) just shows an intermediate robot position in the execution of this action. Particularly, it corresponds to the 20th step.

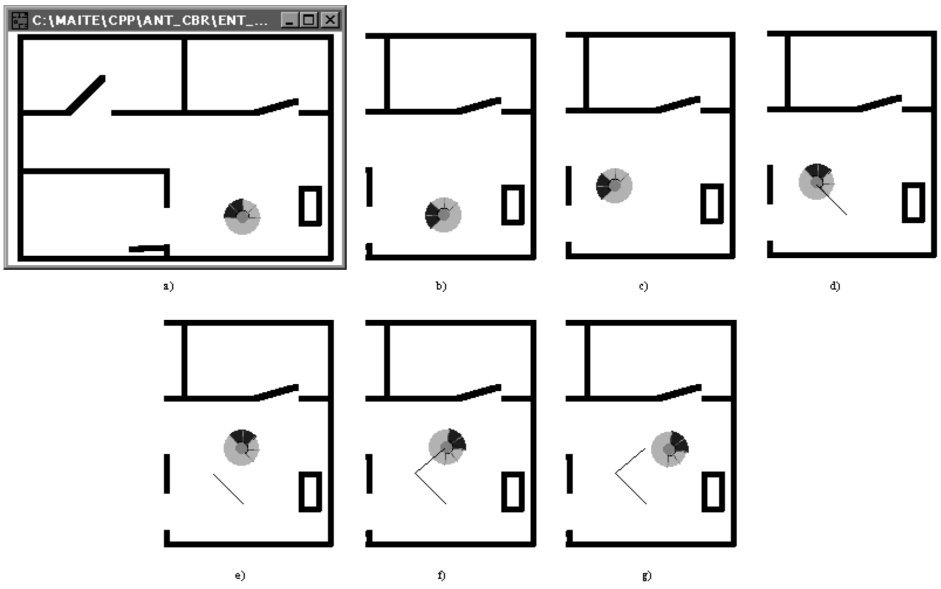


Figure 5.7: Consecutive action executions. Black lines keep track of robot displacements of completed actions. a) Initial position. b) first movement in the first action. c) Last displacement of the first action. d) First movement of the second action (with tack of the first action). e) Second action's last displacement. f) First movement of the third action (showing the trajectory line of the first and second actions). g) Intermediate displacement of the third action.

Simultaneous Robot Movements

The only way by which the user can follow robots' movements is by looking into the interface environment screen. When the user sees the robots moving simultaneously in the environment, what the simulator is actually doing is a constant repetition of a loop that consists of two tasks.

First, the system performs a unitary movement for each of the robots in the environment. This is done sequentially, following the same order in which the robots are distributed in their array. These unitary movements are simple enough to be done in a short period of time. The performance of these movements results in changes at the environment grid as well as variations in the pixel environment representation that is allocated in memory. As we have already seen, these changes are local, and therefore, they do not require an updating of the whole environment representation.

The second task of the loop is an update of the environment screen. The way by which pixel information is stored in memory allows updating the window at once. This implies that the user sees that the screen changes, but her or she cannot see how does it change. In addition, the fact that window updating is done after the movement of all the robots, prevents the user from distinguishing the order in which robots actually moved.

Although snapshots are not the best way of illustrating how the simulator performs robots' simultaneous movements, the next Figure 5.8 tries to depict it by means of some intermediate snapshots. These images correspond to the movements of the three robots from the previous Figure 5.6). Figure 5.8 a) corresponds to the robots' situation after executing 10 steps of their respective actions. Two of them —the ones near the corners— have detected something and they will end their actions. We do not discuss here how do they chose their actions, this is just to illustrate that the user sees the robots moving autonomously and simultaneously. The second image (Figure 5.8 b)), shows the track of these ended actions and new robots' orientations resulted from the execution of their new actions. Figure 5.8 c) represents the immediate execution step following figure b). And finally, after fourteen more steps, we can see in Figure 5.8 d) that the robot in the middle has ended executing its first action.

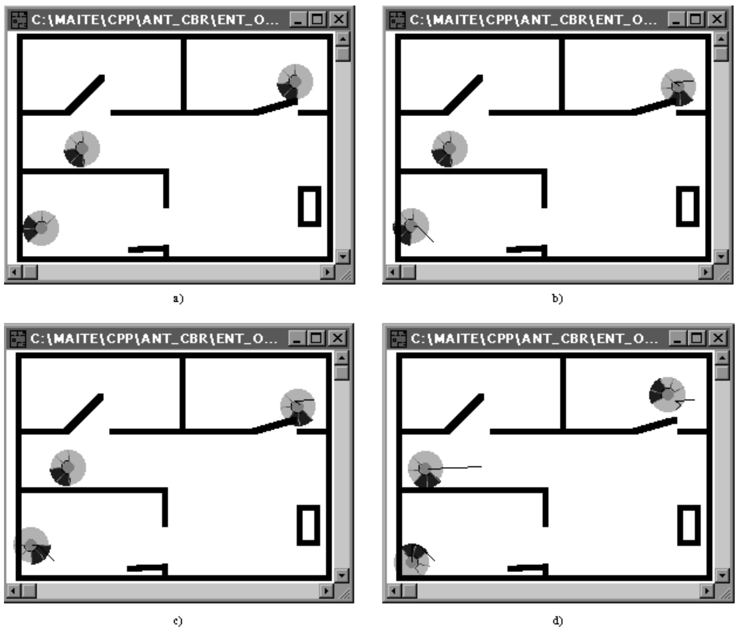


Figure 5.8: Four snapshots of the performance of three robots in the environment. Their initial positions are the ones at Figure 5.6.

5.3.3 Robot Error Implementation

In the first part of this thesis, we analysed the displacement error in a real robot. We concluded that the error increases proportionally with the covered distance, and that this error can be modelled by using two independent Normal distributions. One distribution $N_y(0,1.7)$ models the error in the direction of the displacement and the other distribution $N_x(0,7.3)$ corresponds to the direction that is perpendicular to robot's displacement. Finally, we bounded the errors with probability intervals that cover 95% of the sample.

In order to simulate the robots' errors when they move inside a simulated environment, we first need to generate random errors following these Normal distributions, and second, we have to include these errors in the execution of robots' actions.

Generation of Normally Distributed Errors

C++ mathematical libraries do not provide Normally distributed random generators, they only have available uniform random generators from the standard library stdlib.h. They can be initialised with a fixed seed or with a random value taken from the time of the system (that is done using the time.h library). We have used the uniform random generator random(n) to generate numbers between 0 and n; and this generator has been randomly initialised by calling the randomize() function. From this uniform random distribution we can obtain positive values of a Normal distribution just by considering this uniform distribution as the image of a Normal distribution. Basically, what we do is to consider the random numbers y as the image of the computation of a function over x such that $y=f(x)$ —considering f as the Normal distribution N. And therefore, we compute x as the image of the inverse of the function over y: x=$f^{-1}(y)$.

In this manner, we first consider the Normal distribution, which is defined as:

$$N = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-x^2}{2\sigma^2}}$$

This formula can be rewritten by defining:

$$a = \frac{1}{\sqrt{2\pi\sigma}} \quad \text{and} \quad b = 2\sigma^2$$

So that we obtain a simpler expresion:

$$y = f(x) = a \cdot e^{\frac{-x^2}{b}}$$

Computing now the inverse $f^{-1}(y)$ of this function, we have:

$$x = f^{-1}(y) = \sqrt{-b \cdot \ln\left(\frac{y}{a}\right)}$$

In this manner, from the y values obtained from a uniform random generator, we can have positive x values that are Normally distributed (see Figure 5.9). Since our Normal distributions have both 0 mean (and σ_x=7.3, σ_y=1.7 dispersions), we just have to assign uniformly a sign to the x values in order to cover the whole x-axis. This can be easily done generating again another random number n between a given interval, and then, if n is bigger than half the interval, the x value remains positive. Otherwise, it becomes negative.

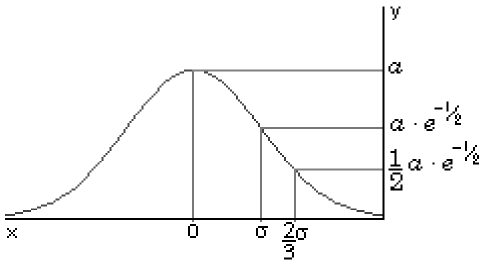


Figure 5.9: Normal distribution N(0,σ).

As we have said, the random number is generated by calling the random(n) function, having as argument the upper bound of the values that the generated number can take. Thus, we have to define this interval of possible values. In order to do it, we must first take into account the restrictions that the function itself imposes over y. On the other hand, y must be positive and non-zero in order to be able to compute the natural logarithm ln function (this is because a is always positive). On the other hand, since b cannot be negative and it appears multiplied by –1, y must be smaller than a —or equal to a— if we want to obtain a positive argument for the squared root. (In fact, we obviously have $x = 0$ for $y = a$).

These previous restrictions are given by the specification of the inverse formula (we could name them as “syntactic” restrictions). In this sense, we still have an additional restriction: a “semantic” restriction. This restriction says that we are modelling an error, and we can not allow x values to tend to infinity just because we have obtained a random value very close to 0. Therefore, we need a lower bound for the y values that must be significantly different from 0. We have chosen this value so that it is half the y value that we have when taking $x = \sigma$. In other words, when $x = \sigma$ we have:

$$y = a \cdot e^{-1/2}$$

and we chose a lower bound y_{lb} for y that is:

$$y_{lb} = \frac{1}{2} a \cdot e^{-1/2}$$

corresponding to the x value:

$$x = \frac{3}{2}\sigma .$$

Summarising, what we actually do is to generate uniform y values:

$$y \in [a \cdot \frac{e^{-1/2}}{2}, a]$$

in order to obtain x values that are Normally distributed:

$$x \in [-\frac{3}{2}\sigma, \frac{3}{2}\sigma]$$

Finally, we have performed several random sample generations with the aim of testing if the computation of random y values is uniform enough to generate a sample of y values that describe an empirical Normal distribution. We have empirically concluded that, considering a sufficient number of elements (up to 150) we obtain empirical distributions that are very close to the Normal distributions $N_x(0,7.3)$ and $N_y(0,1.7)$ that we obtained from the analysis of the sample coming from a real robot. As an example of how close they are, we show the obtained distributions from two generated samples with 300 random values. For the first sample, we considered σ=7.3 and we obtained a distribution

$$N_x(-0.005896, 7.335178)$$

And for the second sample, we considered σ=1.7 and we obtained the following distribution:

$$N_y(-0.119531, 1.729339)$$

Evidently, these mean and deviation values can vary for each generated sample. However, we can see that none of the generated ones where far from the real distributions.

How to Include Errors in the Execution of Actions

Up to this point, we have explained how do we generate the random values that can simulate the errors. Next step is therefore, to include this error in the robots' movement.

The error accumulates with the covered distance. Although there is a direct way of including the error —which consists in computing the random error whenever the robot is redraw in the environment—, the problem of this approach is that the computed error corresponds to an unitary displacement of the robot (which is 2 cm in the simulator). In this manner, if an action determines that a robot must move 1 meter in one particular direction, the error of the position of the robot will be computed 50 times. This can lead to trajectories with two characteristic features. On one hand, since the error distribution has 0 mean, the robot could end in the same place that it would end without considering the error. On the other hand, the trajectory would describe frequent tiny deviations that would lead to a kind of zigzag trajectory. Although this zigzag would be hardly perceptible due to the small variations coming from 2 cm-long displacements, we discard this model because the real robot does not move in this way.

Instead of computing the error for each movement, we decided to introduce the error according to the total distance of the action. In this way, once the system decides what will be the distance and orientation of the next action, instead of assigning them directly, it assigns modified values through an error function that introduces the displacement error. First, this error function computes what should be the destination position if no error were introduced. Afterwards, it generates a random position around the destination position. This random position is generated as we described previously so that it belongs to the interval that estimates the error accumulated during the coverage of the total distance of the action. Finally, once the random position has been generated, the function translates it into new distance and orientation values for the action. Therefore, if the robot executes this action completely, it will end at this random position. Otherwise, it will end in a position containing an error proportional to the covered distance.

In robot movements, turns are executed slightly differently from displacements. This is due to the fact that, for each action, the robot only turns once (at the beginning) whereas it may move forward several times. Moreover, consecutive turning actions —i.e., actions without displacement— imply several turns from different actions. Nevertheless, these differences do not imply a change in the way we assign turning errors, they are included following the same idea than the displacement error. Each time the system decides that the robot must turn α degrees, it assigns a value to the action orientation field that is deviated by an error function. We have already seen that what the real robot has is a constant deviation of two degrees to the left for every ±45-degree turn. And therefore, the orientation deviation depends on the number of degrees turned.

As an example of error inclusion in robots' actions, we return to the Action Execution Figure 5.7. During our comments about that figure, we said that the first action had a distance of 36.82 local units and an orientation value of 44.9 degrees. In fact, the navigation module of the robot had not chosen such values. Indeed, they were the result of adding the error to an action saying the robot to move 74 cm in a left oblique direction. First, a turning deviation was applied to the 45 left degrees, so that the new action orientation was 47°. And afterwards, the system performed several computations in order to include the displacement error. Initiall y, the simulator had to obtain the error bounds in relation to the 74-centimetre displacement: the generated error in the direction perpendicular to the displacement had to belong to the [-2.79, 2.79] interval; whereas the error in the displacement direction had to belong to the [-0.62, 0.62] interval. And in that particular case, the simulator generated the following errors: 2.7 cm for the former interval and -0.4 cm. for the last interval. These errors implied a deviation in the robot orientation of -2.1 degrees as well as a distance reduction of -0.4 cm. In such manner, the action values that were finally assigned are 47 – 2.1 = 44.9 degrees for the orientation field and 36.8 local units for the distance. (Since distance is specified in local units, and since the environment granularity was 2 cm., 36.8 units resulted from the translation of 74 – 0.4 = 73.6 cm).

5.3.4 Robot Information

This subsection details three aspects involving some kind of robot information.

First, the robots' main task is to explore an unknown environment while gathering information about detected features. We call this information partial maps (see Sect. 2.3.1) and the host computer uses them to generate a global map of the environment.

Second, the implementation of robot exploration consists in the execution of a sequence of actions. Since we have already seen that actions do not only contain the movements to perform, but also valuable information about their execution, each robot stores its sequence of actions. We call it action sequence history, and it is used by the robot navigation strategy.

Finally, robots stop exploring when the error accumulated due to its movements becomes bigger than a given threshold. The reason for the existence of this threshold is that information with excessive error is useless. In order to establish the moment at which the robot should stop, it needs to model the error accumulation process.

Partial Map

Section 2.3.1 already described the concept of *partial map* as a representation of a robot exploration trajectory together with detection information. It consists of a sequence of robot positions corresponding to the extremes of rectilinear displacements. For each of these positions the simulator creates what we call an element of the partial map so that the partial map is an array of these elements. A *partial map element* consists of the following fields:

- Robot position.
- Associated Error Rectangle.
- Detection Label.
- Singular Point Label.

We already know that robot movements are action driven. Partial map elements are created for the current robot position whenever a displacement action ends. By displacement action we mean an action that has resulted in a robot displacement. This ending condition is checked upon the state value of the robot action (see 5.3.2 Robot Action Section). The system knows that the action has ended when the state value is bigger than 2. This is so whenever the action has been successfully executed —its state equals 3— or it has been interrupted —which is represented by a 4-valued state.

Together with the current robot position, each partial map element has the error associated to that position. This error increases with robot movements and is approximated by a rectangle specified by means of two corners: the top–left corner and the down-right corner. The following subsection gives details about how the robot generates these error rectangles.

During its exploration trajectory, each time a robot starts following a wall —or obstacle edge— a detection label is associated to its corresponding robot position. This label takes a “right” value if the robot starts following a wall —or obstacle edge— detected by its right sensors. On the contrary, this detection value becomes “left” for left wall following cases.

If it happens to be the case that the robot fails in its wall following action because the wall has ended before leaving it, the robot considers that it has found a singular point. Consequently, a “true” value assignment to the singular point label of the corresponding partial map element reflects such a situation.

Finally, once a robot ends its exploration, it stores its partial map in a file named after the environment and the number of robot it corresponds to. The extension of the partial map file is ‘*.pm’. In the simulator, the memory of the computer is used to simulate the communication between the robot and the host.

Error Modelling

The 2.4.4 Error Propagation Section explains how do we model the accumulation of trajectory errors in the real robots. Basically, we approximate displacement errors by rectangles. For each rectilinear movement, we generate a rectangle going in the same direction of the displacement and we add it to the previous rectangle. The resulting rectangle goes in the direction the robot started with.

Our turning error analysis concluded that the studied robot had a deviation but not a turning error that could be isolated from the displacement error. Deviations do not imply error increase, but just a rotation of the current position respect to the previous turning position.

Although the real robot did not present turning errors, the simulator does accept to parameterise robot-turning errors. They are specified in terms of a segment approximation to the angular error in the direction perpendicular to the displacement. Their units are centimetres and correspond to the length of a segment that is accumulated for each ±45-degree turn and 1 cm displacement (see next Figure 5.10). Furthermore, they are considered to be proportional to the number of turning degrees and different for their sign —that is, right turns can generate different errors than left turns.

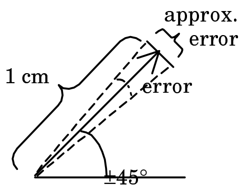


Figure 5.10: Turning error approximation.

In this manner, the robot considers displacement and turning errors whilst generating its partial map. They are utilised to define the dimensions of the error rectangle associated to the current robot position. As before, these dimensions are computed from, on the one hand, the dimensions of the rectangle error at the previous position, and on the other hand, the values that define the last displacement. We reformulate the computation considering the following elements:

- l_p length and w_p width of the previous error rectangle R_p —w_p goes in the robot's initial direction.
- d distance between last and current positions
- α angle difference between last and current orientation, its value results from robot turns. Robot turns are specified by means of two values: α_r right turned degrees and α_l left turned degrees. Consecutive turning actions can be done without displacement, and hence, turned angles with different signs must be considered separatel y in order to compute rectangle error dimensions once the robot moves forward.
- e_d, e_p, e_{rt}, e_{lt}, robot error characteristics. On the one hand, e_d and e_p denote accumulated errors —per covered cm— due to displacement; e_d represents the error in the direction of the displacement, whereas e_p is the error in the direction perpendicular to the displacement. On the other hand, e_{rt} and e_{lt} denote the error accumulation —per covered cm and ±45° turn— in the direction perpendicular to the displacement; r and l subindexes indicate the degree sign: right turns are negative and left turns are positive.

These elements, allow the robot to compute l and w —length and width of the current error rectangle R— applying this formulae:

$$l = l_p + \cos\alpha \cdot (b + c) + \cos(90-\alpha) \cdot a$$
$$w = w_p + \cos\alpha \cdot a + \cos(90-\alpha) \cdot (b + c)$$

where:

$$a = d \cdot e_d$$
$$b = d \cdot e_p$$
$$c = d \cdot (e_{rt} \cdot \alpha_r / 45 + e_{lt} \cdot \alpha_l / 45)$$

Action Sequence History

In previous subsections we have seen that actions contain valuable information about their own execution, and hence, to store this information can help in the robot control. Obviously, since the order of execution of actions determines robot performance, actions are stored following this same sequence. The use of such information can vary depending on the task that make use of it. For example, it can be used to avoid repeating the same action under the same circumstances for more than a maximum number of times, or more generally, to break execution loops with more than one involved action. Moreover, recent action execution can also help in deciding new actions. In this sense, the situation in which last action ended its execution is strongly related to the next action to define. These different aspects will be commented in further detail, when defining the navigation behaviour-based strategy of our robots, in Chapter 6 (Sect. 6.1).

5.3.5 Browsing the Robot Class

Robot class constitutes a complex class that integrates many other classes. This approach is in the Object Oriented direction of code reuse through member objects rather than being in the direction of class heritage. Heritage is usually applied for cases where base classes need to have additional functionality. In this manner, heritage allows to define derived classes without repeating the basic definitions but specifying only the extensions. Another characteristic of heritage is its ability to express a problem in terms of a class hierarchy (Eckel 91). In the definition of a robot class we use member objects instead of derived classes because none of these two characteristics seem to be fundamental for our problem setting. However, this does not mean that instead of having a class that includes many others, we could have used the concept of multiple heritage to define the robot class as a derived class from all the classes that it includes.

The advantage of defining member objects is that we can use pointers to them, so that these objects are composed into the robot class at execution time (in contrast to compilation time). This is usually known as Pointer Pluggable Composition, and it constitutes an efficient way of adding flexibility to the system.

Up to this point, we have already mentioned most of the members of this class. Our aim here is to give a general description of the robot class in order to enlighten the organisation of its members as well as to specify whether they have been included as objects of different classes or as pointers to these objects. Finally, just comment that most of the members of these classes have been defined public in order to allow their access from the robot class.

Data members

We group data members based on what they contain:

• Position: Each robot object has information about its current position as well as its current orientation.

• Characteristics. When creating a robot in the simulator the following features must be defined for it:

- A pointer to the environment that contains the robot;
- Two variables involved in robot's movement: turning probability and left/right turning probability;
- An error characteristics object, which specifies robot's movement errors —displacement errors, turning errors, deviations, etc.—;
- A sensor characteristics object. It sets the size and range of the elements defining the robot: its body radius, range of detection of the presence signal coming from another robot, range of emission of its own presence, distance between the body and the collision detection band, and infrared signal range.

• Members used to draw the robot in the environment. As we have already said, robots have their own representation in local grids that are transferred to the global grid of the environment. In fact, this local representation is stored in three different instances of the local grid class and the robot keeps pointers to them. One of them contains the representation of the robot body and the emission signal. This representation is invariant to the movements, and therefore it is drawn once at the robot's definition and used as base for drawing the robot. It is completely drawn in the other two grid objects, which change whenever the robot turns. In addition, some pointers to several Memory Device Context objects are used to refresh the representation of the robot in the environment screen.

- Sensor and Action objects. Each robot has a Sensor object providing updated sensor readings as well as an Action object, which specifies the current action —that is, the one that the robot is executing.
- Partial map. Previous subsection explained that the robot gathers information during its exploration and stores it in its partial map. The robot object contains a pointer to its corresponding partial map object.
- Historical information. Since action objects contain valuable information about action execution, they are stored in a dynamic array so that it is possible to monitor robots' performance. Again, what robot class actually has is a pointer to an array of actions object.

Methods

Considering the complexity of the robot class, there is no doubt about the importance of its constructor and destructor member functions. They manage the instantiation as well as destruction of all member objects inside the robot class. In addition they control the address assignment of the pointers to other class objects. In fact, robot objects are built in different phases. Initially, when the array of robot is defined, all its members are empty (with 0 values). Afterwards, when the user chooses to add a new robot, a robot object is created based on the definition of robot characteristics, —coming from the user dialog box and default robot values. This object still lacks of pointed composed members, (that is, its pointers to other class objects are null) so that it can be easily copied into the first empty robot at the array of robots. In this manner, its composed members are created —i.e., their memory requirements are allocated— only when the robot is defined in the array. This implies:

- An initialisation of the array of Actions (we will discuss details about arrays later in this subsection),
- Initialisation of the partial map. The first element is the initial robot position with a prefixed error rectangle of 0.2×0.2 cm.
- Definition of the 3 robot local grids, and drawing both the fixed features as well as the robot initial appearance.
- Creation of the objects used to draw the robot in the environment screen. Tbitmap and TmemoryDC.

Once the robot has been completely defined, it only remains to add it to the environment grid and to display it in the screen. If both things are done successfully, the construction ends. Otherwise, if for example, the robot is added into an occupied place or outside the environment limits, the robot object is detached from the array of robots.

In addition to constructor and destructor members, there are other robot class methods that have an important role in the simulation. Most of them have already been mentioned, so we just enumerate them:

- Draw robot and Erase robot
- Read robot sensors
- Introduce random displacement error
- Introduce random turning error
- Execute a step of the current robot action
- Compute current partial map information
- Save partial map file

Some technical aspects regarding the robot drawing method are based on the fact that it uses different Device Context classes defined at the C++ Object Windows Library (OWL). This drawing method uses several object members that determine the context in which graphics are formed (device context members could also be thought as destinations to which graphics are sent). In other words, the drawing method directs its graphical output both to the client area of a window or to an off-screen bitmap just by selecting the appropriate Device Context member of the robot object.

We use two Device Context classes: TClientDC, for drawing in a window's client area; and TMemoryDC, which is used to draw to an off-screen bitmap — gives access to a memory device context. Both are derived from the Device Context base class, and their use involves the creation of graphics objects as TBrush, TPen or TBitmap —a bitmap is a device dependent image that helps in increasing the output speed by holding a copy of the on-screen image. The process of using device context classes to produce graphical output involves four steps:

- First, we need to create or obtain a Device Context object. For example, to create a TclientDC object we do:

  ```
  TclientDC *tempdc = new TclientDC(*current_window_handler);
  ```

  It can also be used to create the TMemoryDC *mdc by doing:

  ```
  mdc = new TMemoryDC (* tempdc);
  ```

- Second, we construct graphic objects as:

  ```
  bitmap = new TBitmap(*tempdc, width_size, height_size);
  TBrush brush(TColor::white); TPen pen (TColor::LtBlue);
  ```

  And afterwards, we select the object into the device context by calling the SelectObject member function:

  ```
  mdc→SelectObject(*bitmap);
  ```

- In the third step we can call one or more member Graphics Device Interface functions for the device context object. For example, in order to set all bits to white in the off-screen bitmap, we can draw a white rectangle having the same size:

  ```
  mdc→FillRect (TRect (0, 0, width_size, height_size), brush);
  ```

  Another example could be drawing a light blue line:

  ```
  mdc→SelectObject(pen);
  mdc→MoveTo (initial_position); mdc→LineTo (ending_position);
  ```

- Finally, we can restore all objects:

  ```
  mdc→RestorePen(); mdc→RestoreBrush();
  delete tempdc;
  ```

  And, if we want to see the draw in the screen, we transfer the off-screen bitmap to the corresponding output device context (by calling the BitBlt member function of the Device Context base class).

Robot class also overloads standard operators such as = or ==. Such operator redefinition is necessary in order to manage robot objects as elements of the array of robots in each environment. Although these standard operators do not constitute an important aspect of the robot class —because many other classes in the simulator overload additional standard operators as +, –, >>, or << when the concepts of aggregation, subtraction or stream transfer apply— they introduce us into the array structures. As its name indicates, the inherited Add function at the array of robots class uses these operators to include a new robot object into the array of robots. Our aim here is not to detail the implementation of such operators but to comment some technical aspects about the array of robots definition. We consider that the array of robots definition is relevant because of the number of other arrays that are implemented in this simulator —as for example, array of actions, walls, or partial map elements—, and hence, we will give an explanation that will be general enough to include all of them.

Borland C++ includes a class library called array.h. This library uses encapsulation and includes several template driven —or template-based— classes of containers. A class template (also called generic class or class generator) lets us define a pattern for class definitions. Basic operations are the same for all classes belonging to each general class definition so that they do not depend on the class member type.

The classes in this array.h library build containers from Abstract Data Types (ADT) and use Fundamental Data Structures (FDS) as underlying structures. The combination of one ADT with a given FDS determines a particular container storage strategy. Borland C++ defines several ADT container classes: array, queue, dequeue, stack, set, bag, and dictionary (with associative keys); as well as several low-level FDS containers: Vector, Btree (binary tree), DoubleList, HashTable, and List. The arrays in the simulator correspond to containers derived from array ADT combined to Vector FDS. The template that defines this combination is:

```
template <class T> class TarrayAsVector;
```

Each template must be instantiated with a particular data type as the type of element that it will hold. Resulting containers provide direct control over the types of objects they store. This is because the compiler takes care of calling the appropriate operators for the contained data type. For example, the simulator defines a type of robot's container called TArrayRobots. This container is an instantiation of the previous TArrayAsVector class template:

```
typedef TArrayAsVector <TRobot> TArrayRobots
```

In this manner, general container functions such as Add(element) or Detach(element) that need to copy, compare or destroy elements, must use the corresponding functions of the given element type. In our case, Add or Detach are called for the instantiated TArrayRobots, and therefore, the compiler will consider the standard operators (=, ==) that have been overloaded for the robot class.

From the definition of TArrayRobots we generate a TRobots class:

```
class TRobots: public TArrayRobots
```

whose constructor is defined in the following way:

```
TRobots(): TArrayRobots(5,0,δ){};
```

to mean that the array is created with an upper bound of 5, a lower bound of 0, and a growth delta of δ. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs, since we have a delta value of δ, the array is expanded —by sufficient multiples of delta— to accommodate the element addition.

Besides standard array operations as Add, Detach or [],TarrayAsVector class has an interesting member function called ForEach. It is specified in the following way:

```
void ForEach( IterFunc iter, void *args );
```

This function creates an internal iterator to execute a given iter function for each element in the array. The args argument lets us pass arbitrary data to this function.

Our simulator calls this ForEach function in order to move all robots inside the array of robots. Getting into more detail, for each environment, the simulator instantiates an object of the TArrayRobots class and several robots are added to this array of robots object. Afterwards, when the user chooses the start robots menu option, the simulator calls the ForEach method for the array of robots corresponding to the active environment. The argument of this call is an IterFunc type of function called move_robot. This call implies one step movement of all robots in the array. As we have already commented, the screen is updated after this call, producing the effect of simultaneous robots' movements.

Up to this point we have explained technical characteristics of the simulator without detailing the navigation policy of the robots. In this manner, we should continue this chapter with a section dedicated to the Behaviour-based architecture for robot exploration as well as a final section dedicated to show the results. Nevertheless, the relevance of the action decision process is significant enough to define a separated chapter, and therefore, we can consider next chapter as a continuation of the present one.

Chapter 6

Behaviour-based Architecture for Robot Exploration

The last paragraph in the previous chapter comments that this sixth chapter is in fact a continuation of the simulation system. It has been separated in order to differentiate the technical aspects from the navigation strategy of the robots. The present chapter describes how robots decide the actions that must be executed in order to perform the exploration task. These decisions are based in a architecture that depends on the co-ordination of different behaviours that are based on If-Then rules. The more suitable way for evaluating the performance of behaviours is to analyse the overall task execution, and therefore, the chapter ends with a detailed section with results (which, obviously, also show results of the previous chapter).

6.1 Architecture

Robot navigation is a complex task that must fulfil abstract goals while interacting with the environment. Divide and conquer strategies proposed by Brooks (Brooks 86), Arkin (Arkin 87) and Payton (Payton 86) have proved to be a good way of coping with the complexity of architectures for navigation because of their decomposition of the problem into small and independent decision-making processes. These processes are called behaviours, and the resulting architecture is know as Behaviour-Based Navigation.

In general, a behaviour-based organisation includes several behaviours, an arbitration strategy and a command fusion operation (Saffiotti 97). Each behaviour fully implements a control policy for one specific sub-task (like following a path, avoiding sensed obstacles, or crossing a door). The arbitration strategy decides which behaviours should be activated depending on the current goal and on the environmental contingencies (Brooks' subsumption architecture hard-wires this arbitration strategy). And finally, when several behaviours are concurrently activated, the command fusion operation combines their results into one effector command.

In our approach, each robot implements a Behaviour-Based navigation strategy that accomplishes the random exploration task. Basically, our architecture is a deterministic finite state automaton in which each state corresponds to an elementary behaviour (see Figure 6.1). These basic behaviours use sensor readings as well as historical information —about previously taken decisions— to determine what are the actions to execute as well as when to switch to other behaviours. In this manner, our approach distributes the arbitration strategy among the behaviours. In Figure 6.1, switching behaviour conditions appear as labels of the arcs, which connect different behaviours.

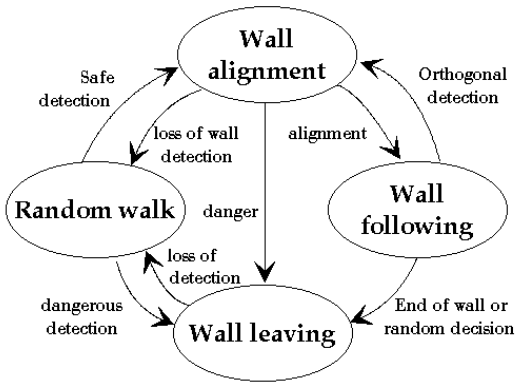


Figure 6.1: Automaton schema of the random exploration strategy.

Each active behaviour provides one action, and only one behaviour is active at a time. Therefore, we do not need to establish any kind of fusion policy. This 'one behaviour active at a time' policy has the advantage of avoiding combination problems for contradictory outcomes, and although it is usually considered as a problem simplification, it is not, because it implies an increase in the complexity of behaviours. In the literature, complex behaviours (Saffiotti 97) are defined to consider multiple objectives, for example, a canonical example is that of following a given path while avoiding unforeseen obstacles in real time. In our approach, elementary behaviours cannot be considered as being complex because they have a single goal. However, they take into account other goals by means of the distribution of the arbitration strategy among them. Considering again the example of path following, our solution would correspond to several separated elementary behaviours: one to follow the path, and one (or more) to avoid obstacles. Their complexity is higher than classical one-goal behaviours: On the one hand, the path following behaviour knows that if the robot encounters something on its way, then the next behaviour that must be active is obstacle avoidance. And on the other hand, the actions taken by the obstacle avoidance take into account the fact that this behaviour has been activated by the path following behaviour.

Our random exploration strategy co-ordinates elementary behaviours in order to cover free space by random changes in robots' direction and by following walls —or obstacle edges— when detected. We briefly describe here each elementary behaviour from the previous Figure 6.1 (next subsection details them):

- *Random walk*: this behaviour is active while the robot is in free-space; that is, it controls the movements of the robot when there are no sensor readings. Random turns and lengths of displacements are decided using the turning and turning direction probabilities. This behaviour is deactivated by any sensor detection.
- *Wall alignment*: this behaviour tries to "align" the robot to what might be a wall (any sensor detection triggers it). By aligning we mean having the robot parallel to the wall —or obstacle edge—, this situation is recognised by the robot when it has sensor information coming from both sensors in one side without any frontal detection.
- *Wall following*: once the robot is aligned to a wall, this behaviour computes the distance the robot will try to cover and controls its displacement by keeping it parallel to the wall.
- *Wall leaving:* the goal of this behaviour is to reorient the robot until there are no sensor readings and then switch into the random walk behaviour.

There are still two more elementary behaviours concerning the communication process between pairs of robots:

- *Presence detection*: any behaviour will switch automatically to this one when a robot detects the presence of another robot. Then, the robot emits a signal and waits for the confirmation from the detected robot so that it can active the data transmission behaviour.
- *Data transmission*: this behaviour consists of the transmission of the information gathered so far, that is, the robot's partial map.

Every behaviour has the same internal architecture. Next Figure 6.2 depicts it. As input, behaviours receive sensor information filtered by a mask, which eliminates irrelevant information for that particular behaviour. Behaviours control the robot navigation by means of a module that contains two sets of rules (If-Then rules that we will see in the next subsection). This module uses these filtered sensor readings together with historical information for providing the output: the action that the robots' effectors will execute or a switch to another behaviour.

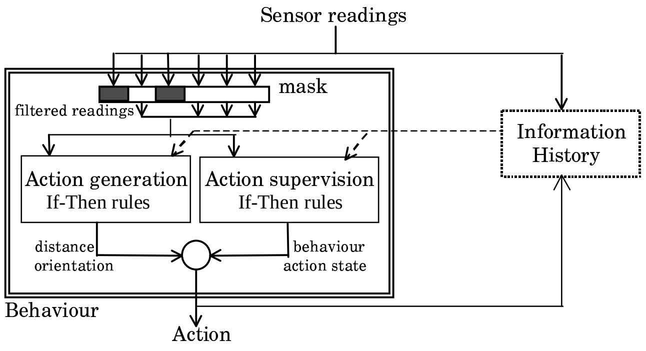


Figure 6.2: Architecture of each behaviour.

Except for the initial robot movement, the navigation control is performed by repetitions of the following sequence:

- Action generation: A set of If-Then rules consider sensor information as well as action history for computing the next action to execute. In general, actions last for several loops, and therefore, they are not necessarily generated at each loop. These rules not only compute new actions but also can modify the current one.
- Action execution. Our simulator executes an action by changing the representation of the robot in the environment. That is, if for example an action sets a certain distance to cover in a given direction, then the simulator moves the robot in this direction for a prefixed unitary distance.
- Sensor update. Once the robot has moved in the environment, then it is necessary to know how the perception of the environment has changed.
- Action supervision. Considering the new sensor data (together with historical information), a set of If-Then rules in the active behaviour check if the current action and behaviour are still valid. Otherwise,

these rules change the state of the action or/and trigger a different behaviour.

The next subsection details action generation and action supervision rules. It is important to notice that these rules do not need to consider every sensor reading in their conditions (as in associative memory matrices). This simplifies the rules although it does not control possible inconsistencies. Moreover, If-Then rules implicitly contain the goal of their corresponding behaviour. Finally, we conclude this subsection by remarking that only one rule is fired for each situation, and therefore, the reaction time is very short and we consider this control navigation to be mainly reactive.

6.2 Description of Exploration Elementary Behaviours

As we have seen, elementary behaviours constitute the navigation module of the robot. When an elementary behaviour is active, it is in charge of generating new actions whenever it is necessary as well as supervising its execution. In addition, each elementary behaviour must define when it has to activate another behaviour so that this new behaviour takes the control for navigation. This subsection details the basic behaviours that the robot uses to explore an unknown environment. For each behaviour, it has been detailed both the If-Then rule set for the definition of new actions as well as the rule set for the supervision of their execution.

Up to this point, we have defined six different elementary behaviours. Nevertheless, the simulator differentiates two kinds of wall following behaviour depending on the robot side that detects the wall. The reason to do it so is the different robot sensors that are involved. Obviously, they are equivalent, though. In this manner, the simulator defines the following ordered set of seven elementary behaviours:

```
{Expl, Align, Leave, R_follow, L_follow, Detect, Trans}
```

Their names have been abbreviated and correspond to Exploration, Alignment, Wall leaving, Right wall following, Left wall following, Robot detection, and Data transmission respectively. They constitute an ordered set just because of implementation simplicity. In this manner, since the Action object includes fields describing the current and previous behaviours (see 5.3.2 Robot Actions Sect.), the fact of establishing an order for their names provides an easy way of specifying conditions over their values. For example, the condition:

```
Action.previous_behaviour < R_follow
```

Is equivalent to:

```
Action.previous_behaviour = Expl    &
```

Action.previous_behaviour = Align &
Action.previous_behaviour = Leave

Before specifying how these elementary behaviours use and assign values to the fields of Action, we abbreviate their names as well, so that they will appear in the sequel as:

Act.dist, which stands for Action.distance,
Act.displ is the abbreviation of Action.displacement,
Act.orient, that equals to Action.orientation,
Act.sens corresponds to Action.Sensors,
Act.behav is equivalent to Action.current_behaviour,
Act.prev_b is the short name for Action.previous_behaviour,
Act.vel is not used here but corresponds to Action.velocity, and
Act.stat, which is the last field of Action: the state.

Each behaviour is in charge of specifying its actions as well as supervising their execution. Most supervision is based on sensor readings. In order to reduce the length of the rules, we abbreviate Sensor names as follows:

Sens.col means the collision detection band,
Sens.pres means the presence detection of another robot,
Sens.l: the infrared sensor placed at the left,
Sens.45l: the oblique infrared sensor on the left,
Sens.f: the frontal infrared sensor,
Sens.45r: the oblique infrared sensor on the right,
Sens.r: the infrared sensor on the right.

Obviously, all distance and orientation assignments that we will detail in the sequel are appropriately disturbed by the error functions explained at Sect. 5.3.3.

6.2.1 Random Walk Behaviour

This elementary behaviour applies four different rules in order to define the next action that the simulator will execute. Since this behaviour is active when the are no robot sensor readings, these actions basically consist of random orientations and distances. The compute_random_distance() function generates random distances based on the robot turning probability and guarantees a lower bound for the resulting values. This is done in the following way:

```
compute_random_distance()
{  minimum = |100–turning_probability |
   n= random (minimum)
   return (minimum + n)
}
```

On the contrary, compute_random_turn() is a function based on the robot left/right turning probability and only generates values belonging to the set {45°, 90°, -45°, -90°}. The orientation value is assigned by calling the turn(α) function, which returns the current robot orientation deviated α degrees.

In the context of this behaviour, since nothing is detected, there are no restrictions on the values of the distance to cover. The case of the orientation is very similar, but not equal, because there are two cases for which a certain orientation is more desirable. Since there are no sensor readings, these cases are characterised by the previous active behaviour: one corresponds to wall leaving whereas the other corresponds to robot communication behaviours (that is, presence detection and data transmission). The task of wall leaving leads the robot into a situation where nothing is detected. Therefore, once it has ended, when the random wall behaviour is activated, it is better not to change the robot orientation again. Otherwise, it is very likely that the robot will find again the wall that it was just leaving. However, this rule has one exception: if the robot was following a wall that ended, then the wall leaving behaviour will have moved the robot along a certain distance from the wall (so that the robot passes the corner safely). In the rules, this case is detected by means of the Act.dist = corner_leaving_distance condition (although in the implementation the inclusion of errors requires a slight modification for this condition). In such a case, the random walk behaviour can freely choose a random direction. Regarding robot communication behaviours as being previous to the random walk behaviour, since a robot detects another robot when this is located at its front left side, it is more appropriate for the robot to turn 180 ° in order to avoid a second immediate meeting.

The four rules that correspond to these situations are:

<u>If</u> Act.prev_b < Detect <u>&</u> Act.prev_b ≠ Leave
<u>Then</u> Act.dist = compute_random_distance(), Act.displ = 0,
Act.orient = turn (compute_random_turn()).

<u>If</u> Act.prev_b = Leave <u>&</u> Act.dist = corner_leaving_distance
<u>Then</u> Act.dist = compute_random_distance(), Act.displ = 0,
Act.orient = turn (compute_random_turn()).

<u>If</u> Act.prev_b = Leave <u>&</u> Act.dist ≠ corner_leaving_distance
<u>Then</u> Act.dist = compute_random_distance(), Act.displ = 0, Act.orient = 0.

<u>If</u> Act.prev_b ≥ Detect
<u>Then</u> Act.dist = compute_random_distance(), Act.displ = 0, Act.orient = turn (180 °).

Once the random walk behaviour has generated the new action to execute, this behaviour is in charge of supervising its performance. The action execution function (see 5.3.2 subsection), which is common for all behaviours, sets the action state to 3 whenever the executed action is successfully ended. Therefore, from the action state value, the behaviour knows when it has to provide another action to the system. In this manner, the behaviour only has to supervise possible contingencies. Random walk behaviour must end and trigger another behaviour whenever something is detected. If it is a collision, then the next active behaviour will be wall leaving. This same wall leaving behaviour is also set when the robot does not face the wall properly. In the rules, this corresponds to a condition that consists in a call to the wall_not_properly_faced() function. This function returns a true value if the frontal sensor gives a near reading and if any of the oblique readings is null or not balanced:

```
wall_not_properly_faced()
{   return (Sens.f = 1  &  (Sens.45r = 0  Or  Sens.45l = 0  Or Sens.45r≠ Sens.45l) }
```

On the contrary, if there is any infrared sensor reading but the condition wall_not_properly_faced() does not hold, then the next behaviour will be wall alignment.

There are three rules that control these cases. Since all of them imply a change in the active behaviour, if any of them can be applied, then the current action is immediately cancelled by assigning a state value of 4.

```
If      Sens.col = 1
Then    Act.behav = Leave,  Act.stat = 4.

If      Sens.wall_not_properly_faced()
Then    Act.behav = Leave,  Act.stat = 4.

If      Sens.IR()  &  ¬Sens.wall_not_properly_faced()
Then    Act.behav = Align,  Act.stat = 4.
```

6.2.2 Wall Alignment Behaviour

The task of the wall alignment behaviour consists in positioning the robot parallel to the wall. In order to accomplish this goal, the alignment behaviour first generates actions that make the robot to face the wall, and then, to turn ±90°. We consider that a robot faces a wall when the following face() method returns a true value:

```
face()
{   return ( Sens.f ≠ 0  &  Sens.45r ≠ 0  &  Sens.45l ≠ 0  &  Sens.45r = Sens.45l) }
```

In other words, we consider that a robot faces a wall when its frontal infrared sensor detects something and, simultaneously, the two oblique sensors have non-zero balanced readings —that is, both are far (equal to 2) or both are near (1).

When the robot faces the wall, the behaviour generates a rule that makes the robot turn ±90°. The turning direction —i.e., the sign of 90 °— is randomly chosen based on the left/right turning probability. This random selection is done due to the lack of information about the wall that the robot is facing. However, the direction does matter when the robot is facing a wall after having followed another wall (that is, when a corner is reached). In such a case, there is only one possible direction to which the robot should turn towards: the direction opposite to the followed wall. There are three rules that reflect these cases:

```
If    Act.prev_b = R_follow
Then  Act.dist = 0,  Act.displ = 0,  Act.orient = turn(90 °)

If    Act.prev_b = L_follow
Then  Act.dist = 0,  Act.displ = 0,  Act.orient = turn(-90 °)

If    Act.prev_b ≠ follow()  &   Sens.face()
Then  Act.dist = 0,  Act.displ = 0,  Act.orient = turn(90 ° * random_sign())
```

The wall alignment behaviour applies the first and second rules when the previous behaviour was wall following. Otherwise (Act.prev_b ≠ follow()), it applies the last rule if, in addition, the robot is facing the wall.

In order to face the wall, the robot must turn depending on the sensor readings it has. We show here the rules for right detection, left detection rules are equivalent but with positive angle signs:

```
If    Act.prev_b ≠ follow()  &  Sens.f = 0  &  Sens.45r ≠ 0
Then  Act.dist = 0,  Act.displ = 0,  Act.orient = turn(-45 °)

If    Act.prev_b ≠ follow()  &  Sens.f = 0  &  Sens.r ≠ 0  & Sens.45r = 0  &  Sens.45l = 0
Then  Act.dist = 0,  Act.displ = 0,  Act.orient = turn(-90 °)
```

This last rule has more conditions in order to give preference to the rules that turn ±45°. In this manner, it is only applied when the first rule does not apply.

If the environment were completely structured and there were no robot errors, the previous wall alignment rules would include most aligning situations. However, this is not our case: in order to face the wall properly, performance errors and oblique walls make necessary to generate small turning actions. The following rules are applied when the robot is safely far away from the wall —that equals to having a far (2) frontal reading— and generate small turns of ±7.5 degrees, which correspond to a sixth of a ±45° turn. The first and second rules apply these turns when the frontal sensor and only one oblique sensor detect the wall. Moreover, the third and forth rules use these small turns when both oblique sensors have non-zero readings but they are unbalanced.

```
If    Act.prev_b ≠ follow() & Sens.f = 2 & Sens.45r = 0 & Sens.45l ≠ 0
Then  Act.dist = 0, Act.displ = 0, Act.orient = turn(7.5 °)
If    Act.prev_b ≠ follow() & Sens.f = 2 & Sens.45r ≠ 0 & Sens.45l = 0
Then  Act.dist = 0, Act.displ = 0, Act.orient = turn(-7.5 °)
If    Act.prev_b ≠ follow() &
      Sens.f = 2 & Sens.45r ≠ 0 & Sens.45l ≠ 0 & Sens.45r > Sens.45l
Then  Act.dist = 0, Act.displ = 0, Act.orient = turn(7.5 °)
If    Act.prev_b ≠ follow() &
      Sens.f = 2 & Sens.45r ≠ 0 & Sens.45l ≠ 0 & Sens.45r < Sens.45l
Then  Act.dist = 0, Act.displ = 0, Act.orient = turn(-7.5 °)
```

These ±7.5° turns help in adjusting the position of the robot with respect to the wall. Nevertheless, they do not always avoid having oscillatory sensor readings. Sensor oscillations occur if, currently, the robot has readings from the frontal and one oblique sensor whereas for the previous action, it was sensing with its frontal sensor and the oblique sensor at the opposite side. To detect such a situation it is necessary to use the sensors of the robot as well as the previous sensor readings —that is, historical information—. We do it by calling the almost_faced() function which can be described as:

```
almost_faced()
{   retun ( (Sens.f ≠ 0 & Sens.45r ≠ 0 & Sens.45l = 0 &
              Previous_Sens.f ≠ 0 & Previous_Sens.45r = 0 & Previous_Sens.45l ≠ 0 ) Or
            (Sens.f ≠ 0 & Sens.45l ≠ 0 & Sens.45r = 0 &
              Previous_Sens.f ≠ 0 & Previous_Sens.45l = 0 & Previous_Sens.45r ≠ 0 ))
}
```

In this manner, the alignment behaviour knows that there is something in front of the robot that gives frontal and oblique readings. Since this situation is reasonably similar to the facing condition, the wall alignment behaviour generates the same random action, as if the (Act.prev_b ≠ follow() & Sens.face()) condition would hold:

```
If    Act.prev_b ≠ follow() & Sens.almost_faced()
Then  Act.dist = 0, Act.displ = 0, Act.orient = turn(90° * random_sign())
```

Finally, the behaviour distinguishes two additional facing situations. On the one hand, when the robot only detects a frontal far detection. And on the other hand, when the robot detects with all its sensors, and the readings are not balanced. For the former situation, the robot is too far and must get closer to the wall. The rule that makes the robot approach the wall by one displacement unit (in local co-ordinates) is as follows:

If Act.prev_b $\neq$ follow() & Sens.f = 2 & Sens.45r = 0 & Sens.45l = 0
Then Act.dist = unit, Act.displ = 0, Act.orient = turn(0°)

Contrarily to this safe detection, simultaneous and unbalanced readings could come from a robot near a problematic corner or any other kind difficult place (notice that the condition includes Sens.f = 1). In such a case, the wall alignment behaviour turns 180° aiming to avoid frontal detection.

If Act.prev_b $\neq$ follow() & Sens.f $\neq$ 0 &
Sens.r $\neq$ 0 & Sens.45r $\neq$ 0 & Sens.l $\neq$ 0 & Sens.45l $\neq$ 0 & Sens.45r $\neq$ Sens.45l
Then Act.dist = 0, Act.displ = 0, Act.orient = turn(180°)

To have so many sensor readings is a problem because this behaviour tries to situate the robot parallel to the wall, and the main condition for being parallel is to do not detect at the front. Nevertheless, when the sensor readings correspond to an unsafe situation, the set of supervision rules forces a switch to the wall leaving behaviour. On the other hand, if the robot has been properly aligned to the wall, then the task of this behaviour has been fulfilled and the next behaviour to activate will be a wall following behaviour. The next rules specify these changes of behaviour cases:

If ¬Sens.IR()
Then Act.behav = Expl, Act.stat = 4.

If Sens.f = 0 & Sens.r $\neq$ 0 & Sens.45l $\neq$ 1
Then Act.behav = R_follow, Act.stat = 4.

If Sens.f = 0 & Sens.l $\neq$ 0 & Sens.45r $\neq$ 1
Then Act.behav = L_follow, Act.stat = 4.

If Sens.col = 1 Or (Sens.f = 1 & ¬Sens.face())
Then Act.behav = Leave, Act.stat = 4.

6.2.3 Wall Following Behaviour

As we have already said, there are two wall following behaviours (R_follow and L_follow) because a robot can follow a wall using its right sensor readings or its left sensor readings. Once the wall alignment behaviour aligns the robot to a wall, the wall following behaviour tries to follow this wall along a random distance that the behaviour computes. This distance is generated only once, when the behaviour is activated, and it is not changed by any subsequent action generated during the rest of the current activation. The task of subsequent actions is to make small changes in the robot direction in order to keep the robot parallel to the wall it is following. The behaviour corrects robot deviations by means of small turns (once again, ±7.5°). Robot deviations are detected when lateral sensors give a near reading or when they lose its detection.

The three rules below implement the right wall following behaviour (the rules that correspond to the left wall following behaviour are equivalent).

```
If    Act.prev_b ≠ R_follow
Then  Act.dist = compute_random_distance(),  Act.displ = 0,  Act.orient = turn(0 °)

If    Act.prev_b = R_follow  &  Sens.45r = 0
Then  Act.orient = turn(-7.5°)

If    Act.prev_b = R_follow  &  (Sens.r = 1  Or  Sens.r =0)
Then  Act.orient = turn( 7.5°)
```

For wall following behaviours, the set of rules in charge of activating other behaviours as well as supervising the execution of the previous actions have to distinguish among several different situations. (Again, we only detail the rules for the right wall following behaviour; the translation to left rules is obvious). From the previous set of rules, the last two rules correct the current action. The behaviour applies them when the robot sensors indicate that the robot is deviating with respect to the wall. In such cases, neither the action nor the behaviour are cancelled, just the action state is set to 2 in order to indicate that the orientation must be corrected. Two rules reflect this situation:

```
If    Sens.r ≠ 0  &  Sens.45r = 1
Then  Act.stat = 2.

If    Sens.f ≠ 1  &  Sens.r ≠ 0  &  Sens.45r = 0
Then  Act.stat = 2.
```

Regarding the activation of other behaviours, most resulting changes end with the activation of the wall leaving behaviour. On the one hand, this can be due to successful ends: whether the covered displacement equals the distance computed for the current action; or the robot finds what we call a singular point (a wall end). On the other hand, the wall following behaviour activates the wall leaving behaviour whenever the robot has a collision, when it looses the wall it was following (i.e., null sensor readings at the corresponding side), or when it finds a dangerous detection that might not leave enough room for the robot (assuming it would continue moving in its current direction). This last situation is considered when there is a close detection —that is, a near sensor reading— at the oblique sensor that is opposite to the wall that has been followed. The corresponding rules are:

```
If    Act.displ ≥ Act.dist
Then  Act.behav = Leave,  Act.stat = 3.

If    Sens.f ≠ 1  &  Sens.r ≠ 0  &  Sens.45l ≠ 1  &  Sens.45r = 0  &  Previous_Sens.45r = 1
Then  Act.behav = Leave,  Act.stat = 4.
```

```
If    Sens.col = 1  Or  (Sens.r = 0  &  Sens.45r = 0)  Or  Sens.45l = 1
Then  Act.behav = Leave,  Act.stat = 4.
```

Finally, it is possible that the robot encounters a perpendicular wall while following the current wall. If this happens and the detection is safe enough, then the next triggered behaviour will be wall alignment so that the robot will try to follow this new wall. The rule that controls such a situation can be defined as:

```
If    Sens.f ≠ 0  &  Sens.45l ≠ 1  &  (Sens.r ≠ 0  Or  Sens.45r ≠ 0)
Then  Act.behav = Align,  Act.stat = 4.
```

6.2.4 Wall Leaving Behaviour

Under normal circumstances, wall leaving behaviour is active when the robot has finished following a wall. Nevertheless, wall leaving is also characterised as the behaviour that is able to face any kind of problem. In this manner, any other behaviour encountering a dangerous situation activates the wall leaving behaviour. We can think of 'dangerous' situations as collisions or sudden near detections. In this sense, 'safe' situations correspond to far or expected detections.

Since turns are safe actions, almost every action that tries to recover from a danger consists in some kind of turning. This behaviour chooses turning angles based on sensor information. Roughly, we can describe angle selection in the following way: when there are more sensor readings at one side of the robot —or they correspond to closer readings— than at the other side, then the robot turns towards the side with less detection. Unfortunately, this policy can not be applied when having balanced sensor readings. In such cases we generate random angles. This gives better performances than to establish a fixed rule because it helps in including variations in the actions, and therefore, avoids action execution loops. Although most of the time turning actions are able to recover the robot from dangerous detections, there are certain situations —such as narrow corridors or difficult corners— for which a change in the robot position can be helpful. Following this idea, we have included into the set of rules an additional rule that —when the sensor readings are not too close— makes the robot move forward one unit.

There are a total number of eleven rules dedicated to control the robot for escaping from problematic situations. We are not going to list them here because, after having shown all the rules for the previous behaviours, the reader can already have an accurate idea of how they look like. What we comment here are the three rules that the wall leaving behaviour uses to control the robot after having followed a wall. (Once again, we show the case of left wall following because the rules for right wall following are equivalent).

<u>If</u> Act.prev_b = L_follow & Act.stat =4 & Sens.col =0 & Sens.f = 0 & Sens.45r $\neq$ 1
<u>Then</u> Act.dist = 50, Act.displ = 0, Act.orient = turn(-15 °)

<u>If</u> Act.prev_b = L_follow & Act.stat = 4 & Sens.col =0 & Sens.f = 0 & Sens.45r = 1
<u>Then</u> Act.dist = 0, Act.displ = 0, Act.orient = turn(-135 °)

<u>If</u> Act.prev_b = L_follow & Act.stat $\neq$ 4 & Sens.col = 0 & Sens.f = 0
<u>Then</u> Act.dist = 0, Act.displ = 0, Act.orient = turn(-90 °)

Basically, the first two rules control the case for which, the action of having followed a wall has been interrupted (state = 4). Under normal circumstances, this happens when the robot has found a singular point (the end of a wall). For those cases, there is nothing dangerous around the robot and it can continue for a distance of 50 units in order to avoid the corner. The -15° turning opens a little the robot trajectory and helps to correct accumulated deviations. On the contrary, under the presence of dangerous detections, the behaviour decides to change completely the robot trajectory, so that it turns –135 degrees. The last rule controls the cases where the Action of following the wall was completed successfully (state = 3). Then, the behaviour just needs to turn 90 degrees towards the direction opposite to the wall it was following.

Finally, we present the rules that supervise the behaviour performance:

<u>If</u> $\neg$Sens.IR() & Sens.col = 0 & (Act.dist $\neq$ 50 Or Act.dist $\leq$ Act.displ)
<u>Then</u> Act.behav = Expl, Act.stat = 3.

<u>If</u> Sens.face()
<u>Then</u> Act.behav = Align, Act.stat = 3.

<u>If</u> Act.dist = 50 && (Sens.col $\neq$ 0 Or Sens.f $\neq$ 0 Or Sens.45r = 1 Or Sens.45l =1)
<u>Then</u> Act.stat = 4.

Basically, these rules switch to the random walk behaviour when the sensor readings are lost, and activate the wall alignment when the robot faces the wall. However, we have said that after a singular point, the wall leaving behaviour still moves the robot forward along 50 units. Obviously, this can only be done if there are no detections on the robot's way. Otherwise, the last rule cancels the action.

6.2.5 Robot Communication

During robot navigation, if it happens to be the case that two robots meet, then they communicate each other their gathered information (that is, their partial maps). Robot communication is a high priority task that is accomplished by the co-ordination of two behaviours: Presence detection and Data transmission. They have not been included into the exploration strategy automaton for different reasons. First, they define a new task, which is not tightly related to exploration. Second, the activation of their behaviours does not depend on the exploration behaviour that is active at a given time. And third, the execution of their actions are neither performed by the same function.

Unfortunately, since only one real robot was available, we were not able to study the real robot communication. The communication process that we have simulated is extremely simple and begins whenever a robot detects the presence of another robot. In such a case, both the current action and behaviour are cancelled so that the communication behaviours get the control of the robot. Afterwards, when the communication finishes, the exploration behaviour recovers the robot control.

Each robot broadcasts a 360° presence signal and detects the presence signal of another robot within a scope of 90° at its front-left side. This presence signal can broadcast two values: '1' is the default value and means the own presence; whilst '5' indicates that the robot has already find another robot and is waiting for establishing communication. Therefore, when a robot detects the presence signal of another robot, the corresponding presence detection field of the sensor readings (that is, Sens.pres) can take two different values (1 or 2 corresponding to '1' and '5' respectively).

Each time a robot detects the presence of another robot with a '1' value in Sens.pres, the presence detection behaviour is triggered. The task of this behaviour is to establish the communication with the detected robot. Thus, it stops the robot, changes its emission of presence into a new '5' signal, and waits for detecting the same change in the other robot. In this manner, when the robot detects a '5' signal, it means that the other robot has also detected him, and then, the behaviour activates the data transmission behaviour. In fact, data transmission behaviour can also be activated directly from any other behaviour, the only condition is to have Sens.pres = 2. This is the case when robots do not detect each other at exactly the same time: for a robot that detects another already waiting for communication, the sensor reading it has is 2.

In our implementation, the partial map communication has been obviated, so that it has been supposed to happen instantaneously. However, if the signal presence is no longer detected during any of the two communication behaviours, the current behaviour will trigger the activation of the exploration behaviour (which will turn 180°). Another reason to cancel the communication is the invariance of the presence signal of the detected robot during the presence detection behaviour. Or, in other words, if the Sens.pres field remains equal to '1', the robot remains stopped for a prefixed period of time, so that when it is completed, the control of the robot goes back to exploration. The presence detection behaviour uses two rules to switch the active behaviour:

```
If     Sens.pres = 2
Then   Act.behav = Trans,  Act.stat = 3,
If     Sens.pres = 0
Then   Act.behav = Expl,  Act.stat = 4.
```

Whereas the data transmission only changes into one behaviour:

```
If     Sens.pres ≠ 2
Then   Act.behav = Expl,  Act.stat = 4.
```

As we have said, the communication actions are special and are executed by a different function. Let us just comment briefly that this function controls the changes of the presence signal that the robot broadcasts. It also updates the robot representation in the environment (the position and orientation of the robot do not change in these behaviours, only the colour of the presence signal) and provides the sensor readings of the robot.

Next Figure 6.3 shows a robot communication example. This figure contains a sequence of snapshots that have been cut in order to show only the most relevant parts of the images. The first image corresponds to the very early step where the robots are still executing the actions of their wall following behaviours. At this step, the supervision rules of the executions detect the presence signal and trigger the presence detection behaviour. In this manner, in the next step, robots compute the new action of the performance detection behaviour that consists of waiting and changing their presence signal into '5' (which corresponds to yellow in the image). When the robot on the left executes its action, the robot on the right has not changed its presence signal yet. On the contrary, since the robot on the right executes its action afterwards, it detects the '5' presence signal and switch its behaviour into data transmission behaviour. The second image shows the result of these executions. In the next step, the robot on the left has finally the sensor reading Sens,pres = 2 and changes its behaviour to data transmission. Then, the right robot computes and executes the data transmission action, and therefore, changes its presence signal (see image number 3). The forth image corresponds to the end of the execution of the data transmission behaviour and a switch to the random walk behaviour. In the fifth image appears how the robots have turned 180 degrees and detect again the wall, so that they switch into the alignment behaviour. Finally, the last image just depicts the situation after several steps.

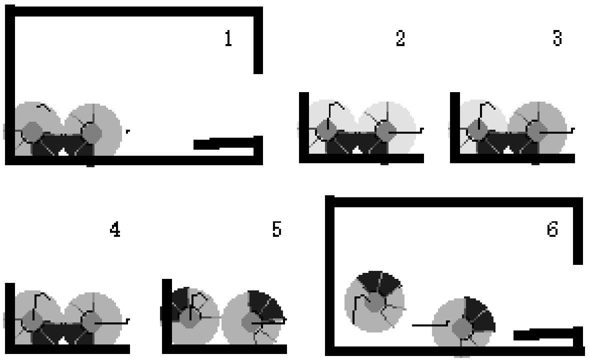


Figure 6.3: Sequence of steps in robot communication

Finally, we show two additional examples in Figure 6.4. In the first example they failed to establish the communication because the robot on the right does not detect the robot on the left. Therefore the left robot waits for a number of steps and finally resumes the exploration. In the case of Figure 6.4 b), the communication is established. The difference with the previous Figure 6.3 is that there, robots detect each other simultaneously, whereas here the robot on the right takes one more step to detect its partner.

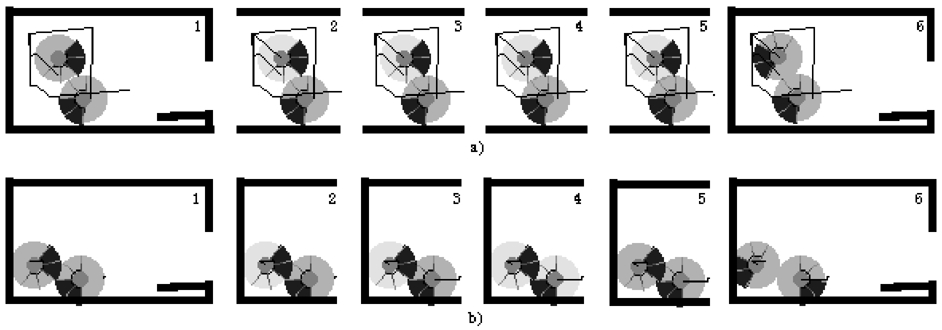


Figure 6.4: Additional examples of robot detection. a) failed communication b) successful communication.

6.3 Meta-level supervision

We distinguish two different levels in the simulated navigation control module of each robot. The lower level is the most complex and corresponds to the level where all the previously seen elementary behaviours co-ordinate themselves in order to generate the actions that the effectors of the robots will execute. The upper level performs a global supervision of the behaviour co-ordination. This meta-level supervision is different to the one performed inside the elementary behaviours. Its input information is the history of actions that have been executed (notice that they contain valuable information about the behaviours that generated them, the sensor readings they had, etc.). The task of this "action history" supervisor is to support high level contingencies. Therefore, under normal circumstances, it does not generate any output. But in special problematic circumstances, the output can be a change in the state of the current action or/and a switch of active behaviour depending on the nature of the problem. These outputs override the respective values in the current action and the behaviour level cannot change them. In this sense, we can think that the action supervision can "inhibit" the behaviour action. Next Figure 6.5 depicts the relation between these two levels in the control module.

Our simulator implements a simplistic version of action history supervision. We use it to detect robot performance loops of a certain number of actions. We just look for coincidences of orientation in a sequence of actions. When they are found, this module sets a different orientation. For example, if two orientations are repeated continuously in the sequence of actions, then the mean orientation is chosen to be the next orientation that the robot must face. In addition, if the distances of the repeated actions are zero and a forward motion is safe enough, then the robot will move forward one unit.

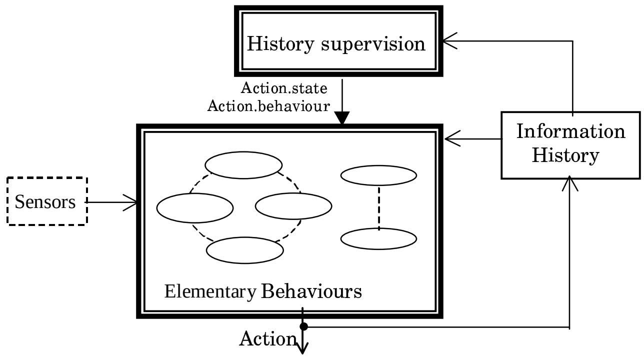


Figure 6.5: Robot navigation Module.

Fortunately, since the generation of actions has a random component, the action history supervisor does not need to act in most cases. Nevertheless, we think it is important to stress the necessity of a global supervisor, and we do not discard the possibility of extending its capabilities.

6.4 Results

Robot navigation for environment exploration has a significant random component (both when generating new actions as well as when introducing execution error). Therefore, since we are not able to show any fixed sequence of robot actions, we present some examples of exploration trajectories. In the simulator, these trajectories appear in the environment screen as black lines connecting the initial and final robot position for each action execution. In the remaining chapters of this second part we will see other examples. These trajectory lines show the general robot performance, such as, wall or obstacle detections, displacements in free space or wall followings. However, they are not able to clearly reflect consecutive turns or small displacements. This section is meant to detail the kind of movements that the robot performs when facing specific situations. Usually, the user of the simulator starts the robots and let them move in the environment until they stop due to an excessive accumulation of error. Nevertheless, if the user wants to pay attention in robots movements, then he or she can move them step by step or every each 10 steps. One step in the simulator interface corresponds to the execution of one step for each of the robots moving in the environment. Using this simulator capability we have been able to take as many snapshots as we need. In the sequel we show several snapshots that we consider to be relevant for illustrating robots' performance.

The next two figures (Figure 6.6 and its continuation Figure 6.7) show the trajectories of two robots. They have been included into one of the environments that we saw at Figure 5.1 (whose granularity is 2, its dimensions are 5.7m $\times$ 4m, and its origin is located at the bottom-left corner). The robots have been added with the default error characteristics and different probabilities and positions. In the sequel we will refer to the robot near the origin as robot number 1, and the robot above will be robot number 2. Their turning probabilities are 0.5 and 0.2 respectively, and, following the same order, their left/right turning probabilities are 0.6 and 0.4. The first image in Figure 6.6 shows robots 1 and 2 in their initial positions, which correspond to (50, 40) and (45, 100).

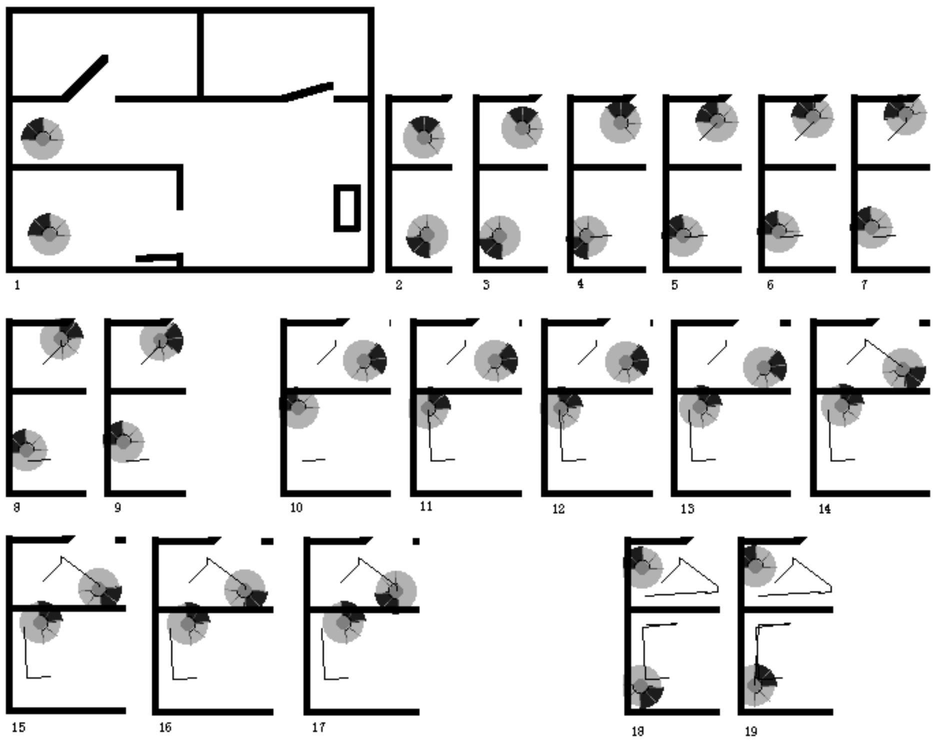


Figure 6.6: Example of robots' trajectories.

The first movement of the robots is controlled by the random walk behaviour. This behaviour chooses randomly the distance and orientation to follow. The second image shows the first step of the execution of the computed action. Robots move in this same orientation for more than ten steps, until they detect a wall and switch behaviour into the wall alignment behaviour. The third image in the figure presents the moment in which robot number 1 detects the wall, whereas the wall detection of robot number 2 appears in the forth image. By then, robot 1 has approached the wall in order to face it. The fifth image shows the first wall alignment action for robot 2 and the last one for robot 1. Robot 2 has detected the wall with its oblique left sensor, and then turns 45° in order to face the wall. On the other hand, robot 1 was facing the wall properly so it randomly decides to turn -90° and switches into the wall following behaviour. In fact, robot 1 keeps following the wall from image 6 to image 10. These images depict the problems that robot 2 has when trying to face the corner that a half-opened door defines with the wall. In the sixth image robot 2 has approached the wall in order to detect it with both oblique sensors. However, the right oblique sensor does not detect the wall, and we can see in number seven how the wall alignment behaviour has made the robot to turn 7.5° twice. This movement does not bring the robot to an oblique right detection but, on the contrary, it results in a near frontal detection that is considered to be dangerous. Therefore, the supervision rules of the wall alignment behaviour switch into the wall leaving behaviour. Image 8 presents the first action of wall leaving, which consists in a -90° turn. The following image shows that this action does not prevent the robot from detecting the wall, and therefore, a total number of 6 turning actions (of –7.5 ° each) and a displacement are executed before the situation shown in the 9th image is reached.

Still in Figure 6.6, the tenth image represents the robots' situation after executing 24 additional steps. By then, robot 1 has detected a perpendicular wall and robot 2 is far away from the corner due to a new random walk action. The eleventh image displays how wall alignment behaviour makes robot 1 to turn –90 degrees. After the execution of this turning action, the left wall following behaviour becomes the next active behaviour. From image 12th to 17th robot 1 keeps executing the wall following behaviour. It corrects its orientation by 7.5° at image 12 in order to detect with the oblique left sensor and by -7.5° after image 17, when the left sensor has a near reading. Regarding robot 2, these previous images show how it first finds a new wall in the 13th image, it turns -45° in image 14th, gets closer to the wall by two displacement units in image 15th and, since its left oblique sensor does not detect the wall, it turns -7.5° in image 16th. This corresponds to an almost faced condition so that the last action that the wall alignment behaviour generates is a -90° turn. Afterwards, the left wall following controls robot 2.

Wall following ends when the robot finds a perpendicular wall, when it looses the wall, or simply when the distance that was randomly obtained for its action has been successfully covered. Image number ni neteen depicts this situation for robot 1. Then, the robot leaves the wall it was fol lowing by turning 90 degrees to its left (see the result of the turning in image number 20).

In the second figure continuing with this example of robots' trajectories (Figure 6.7), we concentrate now in robot number 2. In the 20 th image, this robot is under the control of the left wall following behaviour. In fact, this image shows the last time that the left oblique sensor detects the wall normally. In the next step (image 21), this sensor looses the reading, and therefore, the behaviour corrects by 7.5 degrees the orientation of the current wall following action. Image number 22 reflects this situation. Unfortunately, this turn is not enough, and when the robot moves forward it looses again its oblique detection as we can see in the 23 rd image. Once again, wall following behaviour corrects the robot orientation and, although now the robot has readings coming from both left sensors, it is getting dangerously close to the wall (see 24th and 25th images). As a consequence, the supervision forces a switch to the wall leaving behaviour. Finally, image number 26 displays robot 2 leaving the wall.

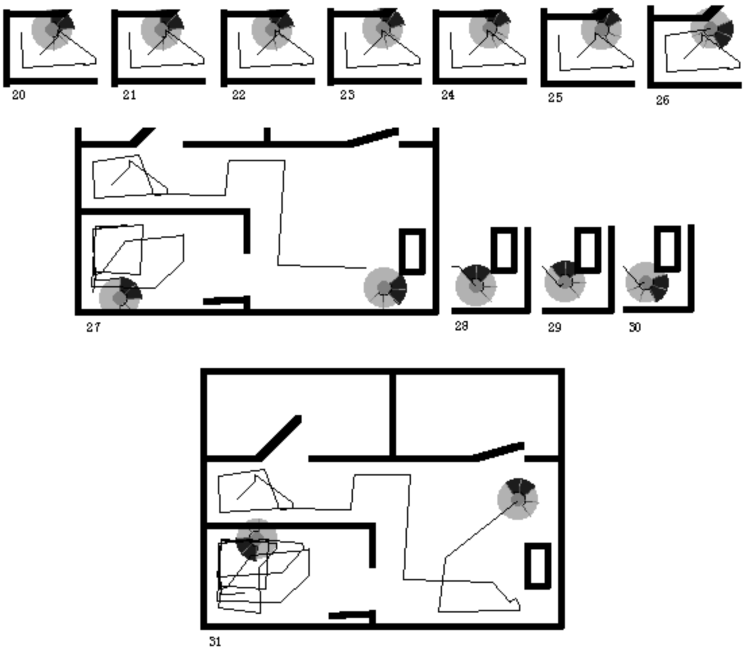


Figure 6.7: Example of robot's trajectories (continuation of Figure 6.6).

After executing a large number of steps, robot number 1 is still inside the bottom-left room in the environment, whereas robot 2 has moved towards the bottom-right corner. In Figure 6.7, the image number 27 contains both robots' trajectories in the explored environment. We have taken this snapshot in order to present another case of corner approaching. Initially, robot 2 is controlled by the random walk behaviour and it detects the object near the corner with its left sensor. Then, the supervision rules forces the active behaviour to be wall alignment so the robot turns 90 degrees and it detects the obstacle with its frontal sensor (see image 28). This behaviour tries to face the obstacle by approaching the robot. Unfortunately, corners can never be faced because the frontal sensor becomes near before any oblique sensor detects anything (image 29 represents this situation). In this manner, when the frontal sensor gives a near reading, the supervision in the wall alignment behaviour cancels itself and activates the wall leaving behaviour. In this case, a -90° turn is enough to stop detecting the obstacle.

Finally, the last snapshot displays the overall robots' performance. At the beginning of this example, we said that robot 2 has a smaller turning probability than robot 1. We can appreciate here, that robot 2 covers longer distances than robot 1 and it is not only due to the fact that robot 1 is in a smaller room. We can see several straight forward movements of robot 1 that have been finished just because of the random distance that was computed for the corresponding actions.

In the previous example we could see the problems that robot 2 had in following the wall when approaching the half-opened door. Moreover, the robot could neither face the corner that the door makes with the wall. In fact, this 45-degree-opened door represents the worse case of oblique edge that a robot can find. This is because the angle difference is too large to be followed as a single wall but too small to detect the end of the wall (a singular point). And the same happens when trying to face the corner. We can easily see this just comparing the difference in terms of number of required actions of facing a straight wall (180°) or a 270° corner with this 225° corner. However, this problem only appears when finding the corner, not the door itself. Neither the action generation rules nor behaviour switching rules take into consideration the angle of the wall. They just consider relative positions between the sensors and the wall and, therefore, they are applied in the same way for oblique walls. In order to illustrate how robots behave in relation to oblique walls (in this case, a door can be thought as a wall) we see in the sequel several examples of specific situations.

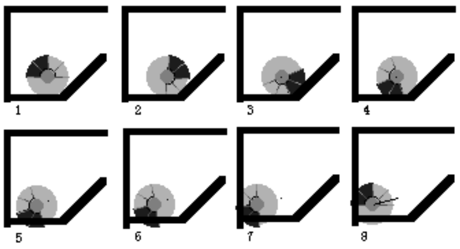


Figure 6.8: View of a robot facing a 135° corner.

Figure 6.8 presents an example of a robot facing the same half-opened door from the opposite side. For this case, the wall alignment behaviour succeeds in, first, facing the corner (by turning -45 °) and afterwards, turning –90 degrees. As usual, the control is passed on the left wall following behaviour. Obviously, wall following does not start with the most convenient orientation, but this is corrected in the subsequent steps. The last images in the figure demonstrate how the perpendicular wall is normally faced.

The next example presents a wall following situation where the followed wall becomes oblique at a point. The first sequence in Figure 6.9 proves that such kind of wall can be successfully followed. Unfortunately, the track is only a line between the initial and final points of an action and, for this particular situation, it does not describe exactly the trajectory of the robot. In this case, the distance of the action was covered before reaching the door end. But this situation does not present any problem for the wall leaving behaviour, which just had to turn 90 degrees to avoid any detection.

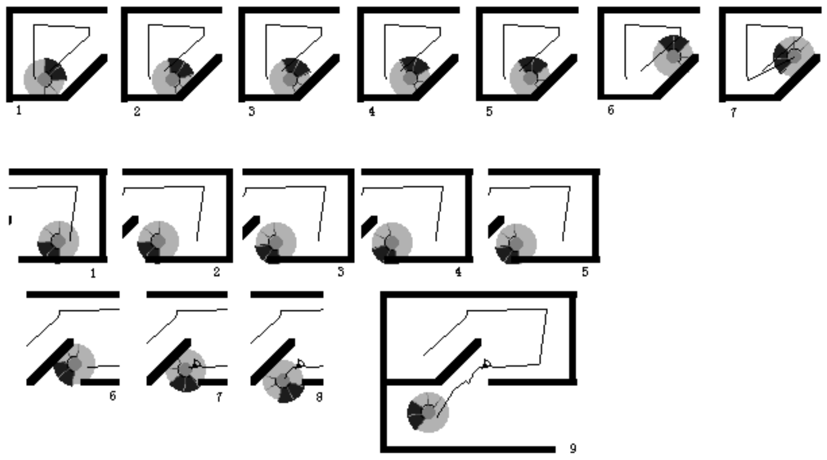


Figure 6.9: Two more examples of robots' performance. First sequence depicts how a robot follows an oblique wall. In the second sequence, the robot finds a singular point (a wall end).

The second part of Figure 6.9 illustrates how the robot detects singular points (that is, the end of a wall). They are easily identified by a certain sequence of readings in the oblique sensor. This sequence differs from a regular wall following in that sensor readings do not change gradually but from near to non-detection. This can be interpreted as a sharp shape, and the robot stores this position as singular point. The behaviour that is active afterwards is wall leaving and it tries to escape from the corner by moving forward along a prefixed distance. However in this particular case, this is not possible and the current behaviour has to force the robot to go through the door, a situation that is rather complex and requires quite a lot of steps (notice the "knot" in the trajectory).

Finally, our last example is devoted to demonstrate that a singular point can be also identified in the case of an oblique wall or obstacle. The sequence of readings is exactly the same, and again, the wall leaving behaviour forces the robot to try to escape from the corner. In this case the situation is more favourable than the previous one because of the lack of near detections obstructing the robots' way. The following Figure 6.10 shows how easily the robot turns around the door.

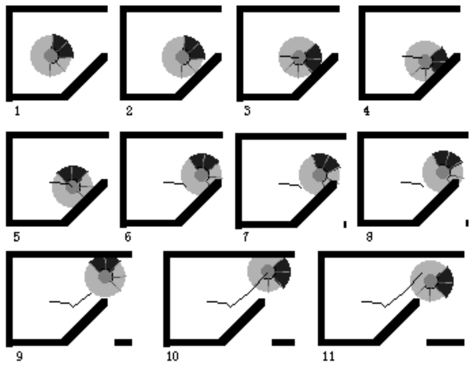
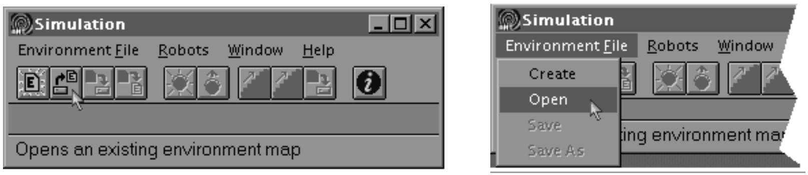


Figure 6.10: Example of a robot following an oblique door, detecting its end and turning around it.

6.4.1 A Complete Example: from the Opening of an Environment to Saving its Partial Map

With the aim of illustrating the whole simulated exploration process, this last subsection presents an additional example that includes interface aspects.

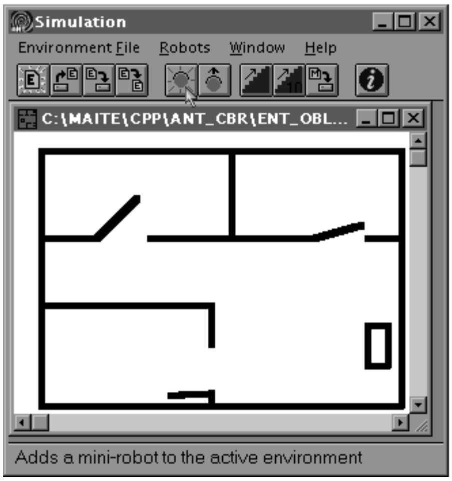
Initially, the simulator application only allows the user to open an existing environment or create a new one. This can be done by means of the buttons in the toolbar or the corresponding options in the 'Environment File' menu. As we can see in the following Figure 6.11 all remaining options appear disabled.



Opens an existing environment map

Figure 6.11: Simulator application.

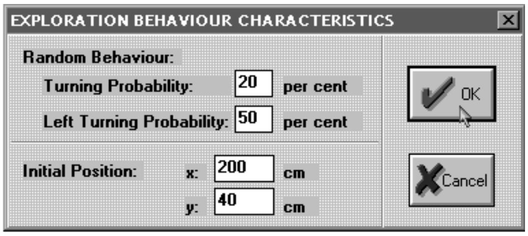
For this example the user has chosen the option of opening an existing environment (which corresponds to the one pointed by the cursor in the previous figure). This selection results in the same window as any standard open file dialog, it lists the names of the environment files that are available (see Figure 5.1). In this manner, the user opens an environment so that its screen becomes active, (notice that the environment is the same than the one in previous examples). The next Figure 6.12 depicts how the user can afterwards add a new robot by pressing the fifth button. (Again, it has an equivalent menu option).



Adds a mini-robot to the active environment

Figure 6.12: Opened environment in the simulator.

We already saw in Figure 5.2 how the robot characteristics dialog allows the user to change the error default values. Since, we have not changed them for all the presented simulations, we just include here a figure containing the dialog used to define the probabilities and initial position of the new robot (Figure 6.13).


EXPLORATION BEHAVIOUR CHARACTERISTICS

Random Behaviour:

Turning Probability: 20 per cent

Left Turning Probability: 50 per cent

Initial Position: x: 200 cm

y: 40 cm

Figure 6.13: Robot characteristics dialog.

The user can repeat this process of adding exploratory robots. The only restriction concerns their initial positions. On the one hand they must belong to the environment co-ordinates. And on the other hand, robot bodies can not be placed in physically occupied positions (i.e., locations of walls, obstacles or other robots' bodies). (The system will punctually indicate both circumstances).

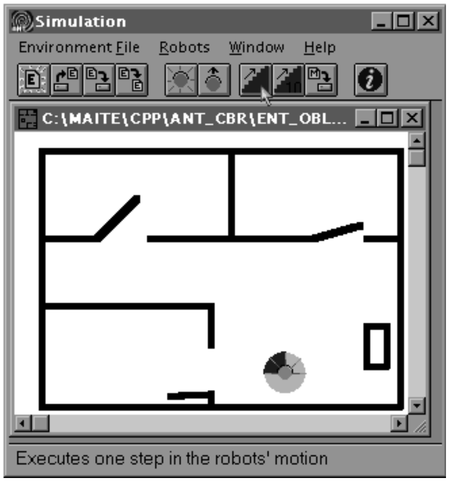

Simulation
Environment File
Robots
Window
Help
C:\MAITE\CPP\ANT_CBR\ENT_OBL...
Executes one step in the robots' motion

Figure 6.14: Starting the robot exploration.

Once the robot is positioned in the environment, several options allow the user to start the exploration (In case there of several robots, they will start their exploration simultaneously). If the user presses the sixth toolbar button (or its equivalent 'Start Robots' option from the 'Robots' menu), all the robots will explore the environment until they accumulate an error that exceeds the threshold value. Since the error is approximated by rectangles, we consider that a rectangle exceeds this threshold when one of its sides exceeds it. In all the examples shown here, this threshold value has been set to 1 metre.

The user can select a different option to start the robots and move them step by step. The simulator offers this alternative option to allow the user to supervise their performance. In the previous Figure 6.14, the mouse cursor points to the toolbar button that executes one step of robots' motion. The button next to it is meant to continuously move the robots during 10 steps. Both buttons (and their equivalent menu options) can be alternatively pressed so that the user can focus in specific robots' actions.

We have been using these two latter options to take snapshots of the exploration performed by the robot included before. Once more, snapshots have been cut and numbered. The following Figure 6.15 depicts all these images. From the previous examples we already know how behaviours coordinate themselves. Here, we are interested in showing the robot trajectory as well as its wall following. Except for the last image, all the presented images focus on the first two wall followings, whose steps have been executed between images 2 and 3, and from 9 to 10.

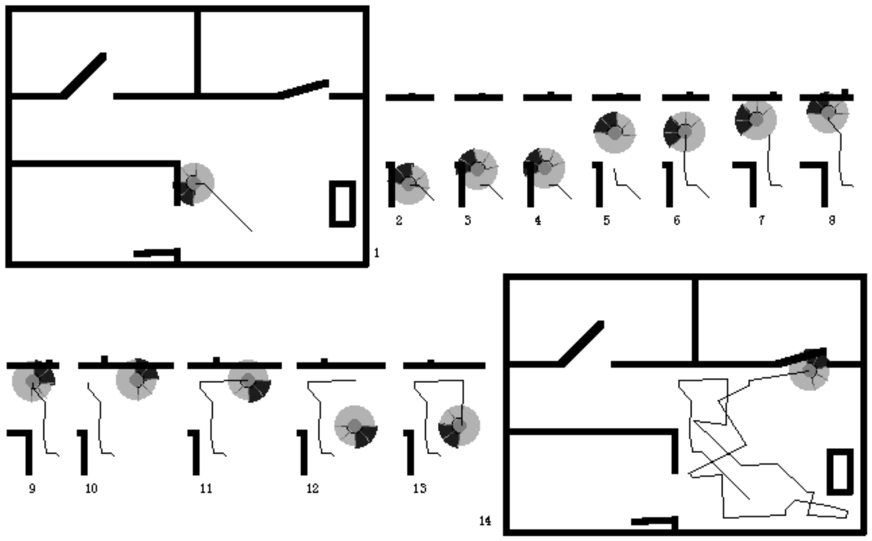


Figure 6.15: Robot exploration.

Robot's Partial Map

When moving the robots step by step, the 'Save Partial Maps' button generates a file for the partial map of each robot in the environment. As subsection 5.3.4 indicates, a partial map contains the robot trajectory together with wall following information. For this example, we saved the partial map of the robot after the thirteenth image. The partial map file is a binary file whose name is built from the first 5 characters of the environment's name and the position of the robot in the array of robots. Binary partial map files have a '*.pm' extension. The system also stores the same partial map information coming from exploration in a text file. This file has the same name but has an '*.exp' extension. We show the contents of this ent_o_0.exp file in the following Table 6.1. This table contains 6 columns. The first column just indicates the row number and has been included for the sake of clarity. (The first information in the partial map is the number of rows, though). The second column is the most important, and contains the position of the robot. As the rest of positions in the partial map, this position is specified in local co-ordinates (so that the origin is the bottom-left corner in the environment). The third and fourth columns define the estimated error rectangle that is associated with each robot position. The rectangle error is always initialised with a fixed 0.1 cm size so that the initial position of every partial map always has this associated rectangle error. And finally, wall following information is stored in the last two columns: s.pt. refers to singular points and wall to wall following.

num	robot position	upper-left error	bottom-right error	s.pt.	wall
1	(200, 40)	(199.9, 40.1)	(200.1, 39.9)	no	no
2	(164.703, 75.4141)	(160.462, 79.6464)	(168.944, 71.1819)	no	no
3	(161.707, 75.271)	(157.386, 79.7911)	(166.027 70.7509)	no	no
4	(160.709, 75.207)	(156.360, 79.8233)	(165.057 70.5907)	no	no
5	(159.711, 75.1478)	(155.335, 79.8602)	(164.087 70.4353)	no	no
6	(157.713, 75.0564)	(153.284, 79.9607)	(162.141 70.1522)	no	left
7	(156.185, 87.9029)	(150.489, 93.3479)	(161.882 82.4579)	yes	no
8	(157.373, 112.875)	(149.279, 118.982)	(165.468 106.767)	no	no
9	(147.788, 123.079)	(138.513, 130.321)	(157.063 115.836)	no	no
10	(147.960, 125.071)	(138.492, 132.374)	(157.428 117.769)	no	no
11	(148.022, 126.069)	(138.458, 133.4)	(157.586 118.739)	no	no
12	(148.081, 128.068)	(138.325, 135.448)	(157.836 120.688)	no	left
13	(184.015, 129.055)	(173.284, 139.892)	(194.745 118.217)	no	no
14	(183.195, 95.0644)	(169.217, 106.728)	(197.173 83.4011)	no	no

Table 6.1: ent_o_0.exp partial map.

Now, from the information on the partial map and the previous image we can easily specify the actions that the robot has performed until the situation in image number13.

- Initially, the robot has moved from its initial position (1st row) until a position (2nd row) where it has detected a wall. This displacement corresponds to the first oblique black line in the first image of Figure 6.15.
- This detection forces the wall alignment behaviour to take the control of the robot. Under this behaviour, several actions are executed. Their final positions correspond to the 3rd, 4th, 5th, and 6th rows in the table. The first image in Figure 6.15 shows the robot in the 5th position, whereas last position corresponds to the second image. Is in this same position where the left wall following behaviour becomes active, and therefore, last column in the 6th row takes the value left.
- The robot follows the wall until it finds its end. Images 3 and 4 depict the detection of the singular point. This is recorded in the partial map by including a new row (number 7) and assigning yes to the s.pt. column.

- The next active behaviour is wall leaving, which makes the robot to escape from the corner. Under this behaviour, the robot reaches the position in the 8th row and 5th image.
- When nothing is detected, the robot control returns to the random walk behaviour. Images 6 and 7 shows how does the robot moves until a new wall is detected. This position is reflected in the 9th partial map entree. Again, the wall alignment behaviour causes robot movements that appear in images 8 and 9 as well as in the partial map (see 10, 11, and 12 rows).
- The start of the second left wall following is indicated in the wall column of the 12th row. And ends in row 13 (see also image 10). Once the wall following distance has been covered, the wall leaving behaviour generates a -90° turning action. The 11th image reflects the new robot's orientation. The partial map does not reflect it because turnings do not change robot's position.
- Finally, images number 12 and 13 depict a random walk action that ends in the last position of the partial map.

Up to this point, the user will probably be as tired of the example as the reader. A quick way of finishing the simulation is by pressing the 'Start Robots' toolbar button. This finishes the step by step simulation and moves the robot continuously until the error exceeds its threshold. Last image in Figure 6.15 depicts the complete simulation.

Whenever the robot ends its simulation, it stores its partial map (within the same text and binary files). The text map file contains 44 entrees. We just copy here the ones that correspond to wall followings. The binary file contains the same structure than the real robots' information. This structure has robot positions defined as integers, and this is why we have rounded its co-ordinates in Table 6.2.

num.	robot position	upper-left error	bottom-right error	s.pt.	wall
1	(200, 40)	(199.9, 40.1)	(200.1, 39.9)	no	no
⋮					
6	(158, 75)	(153.284, 79.9607)	(162.141, 70.1522)	no	left
7	(156, 88)	(150.489, 93.3479)	(161.882, 82.4579)	yes	no
⋮					
12	(148, 128)	(138.325, 135.448)	(157.836, 120.688)	no	left
13	(184, 129)	(173.284, 139.892)	(194.745, 118.217)	no	no
⋮					
24	(243, 28)	(219.644, 52.5041)	(266.967, 3.60341)	no	right
25	(272, 26)	(247.782, 53.6432)	(296.718, -0.835667)	no	right

26	(273, 31)	(247.672, 58.7796)	(297.576 3.9849)	no	right
27	(259, 35)	(233.356, 63.3308)	(284.884 5.85073)	yes	no
⋮					
32	(227, 28)	(198.776, 60.4271)	(254.464 -3.86025)	no	left
33	(181, 26)	(151.597, 62.3778)	(209.775 -10.7455)	no	no
⋮					
43	(200, 128)	(158.77, 177.261)	(241.762, 78.3895)	no	left
44	(245, 135)	(201.75, 188.422)	(289.108, 80.5913)	no	no

Table 6.2: Wall following entrees in the partial map.

We can compare this wall following partial map entrees with the trajectory in the last image of the previous Figure 6.15. First there are two left wall followings, from which one has detected a singular point. Afterwards there are three consecutive right wall followings in the bottom-right corner of the environment (the last one also indicates the existence of a singular point). And the last two detected walls have been followed from the left side of the robot. The last one corresponds to the semi-opened door.

Wall Coverage

In the partial map there is additional information:

- Number of stored positions = 44
- Number of followed walls = 7
- Number of detected singular points = 2
- Covered wall distance = 188 environment units.

This last data corresponds to the total length of followed walls. It is measured in units that depend on the environment granularity. Therefore, since the environment in the example has granularity 2, walls have been followed along 188 $\times$ 2 = 376 cm. Taking into account that the total length of walls in this environment is 2938.45 cm, we can see that the percentage of wall coverage is 12.8%.

This result considers all the walls in the environment. Nevertheless, some of them are unreachable for the robots. For example, since the door of the upper-right room is almost closed, two of its walls cannot possible be detected. This decreases the total followable wall length by 530 cm. In that case, the percentage would be 15.6% of reachable wall coverage. The following chapters will further study wall coverage.

Chapter 7

Map Representation

The last chapter details the way in which robots explore an unknown environment. In addition, it specifies how do they acquire their partial maps from their trajectories in free space and wall followings. Once the robots have completed their exploration, they communicate their partial maps to a host computer. The present chapter focuses on how the host computer uses the robots' partial maps in order to generate a global map. This global map constitutes a compilation of all the followed walls and covered free-space specified in the received partial maps.

Odometry errors generate imprecise information about the position of detected environment features and free-space locations. The host computer makes use of Possibility Theory (Dubois 88) to model the imprecision of the information. Given some vague information, Possibilistic Logic evaluates the truth of crisp propositions. In our case, such propositions are:

- "The position of the robot corresponds to the (x, y) free-space position", and
- "The robot is following a wall at (x, y)"

Where x and y are crisp numbers.

For the former proposition, we assume that the robot is indeed in free-space, and we evaluate the precision of its crisp position (x, y). Equivalently, the later proposition assumes that the robot is actually following a wall, although the location of this wall is not precisely known.

The host computer evaluates the truth of these propositions on the basis of odometry errors from which it derives degrees of possibility Π and necessity N. Therefore, the resulting global map models the environment in terms of degrees of possibility and necessity of the position of detected walls and obstacles.

In this chapter we will see that our approach consists in generating a global map that discretizes the environment in a bidimensional grid. Each cell (*i, j*) in the resulting grid map contains two values: the degree of possibility and the degree of necessity of the presence of obstacles. Initially, before any exploration has taken place, the possibility that a cell at co-ordinates (*i, j*) is occupied by a wall is 1 (that is, $\Pi_{ij}(wall) = 1$), and its corresponding necessity value is $N_{ij}(wall) = 0$. These initial values represent ignorance and are intuitively interpreted as 'although it is completely possible that there is a wall in any position, there is no certainty at all about it'. Afterwards, sensor detection information yields to positive values of Necessity of wall or obstacle, whereas information about free space gives Possibility values smaller than one. In this manner, the map generation process consists in a combination of new possibility and necessity values — derived from received partial maps— with the ones at the current global map. The host implements the map generation process incrementally, by adding new partial maps when robots communicate them. Incremental computation allows the host to generate the global map with the information available at any moment. Therefore, the host does not need to wait for all the robots to successfully end their exploration.

7.1 Possibility Theory

Possibilistic Logic is a particular case of Fuzzy Logic (Godo 93). Given some vague information, Possibilistic Logic evaluates the truth of crisp propositions whereas Fuzzy Logic evaluates the truth of vague propositions. In our case, propositions are of the kind "the position of the robot corresponds to the (x, y) free-space position" and "the robot is following a wall at (x, y)" where x and y are crisp numbers. (Note that the opposite of the Possibilistic Logic is the Multivalued Logic, which assigns crisp values to the truth of vague propositions).

Possibility Theory defines a possibility distribution based on the fuzzy sets. In Section 3.1.1 we represented a fuzzy set by its membership function:

$$\mu_A : U \rightarrow [0,1]$$

Were A is a vague predicate over the values from a universe *U*. This membership function represents the predicate by assigning membership degrees to the elements of the universe. Possibility Theory states that the membership function μ_A equals the possibility distribution π_A. This π_A distribution is the set of possible values of a variable over the universe *U* given a predicate A, with the understanding that possibility is a matter of degree.

In our case, given a detected segment $S = [(x_1,y_1), (x_2,y_2)]$ and its error Area_e, we defined its associated fuzzy set F_s as a membership function μ_W that assigns, to any point (x,y), its corresponding membership degree to a real wall. Now, the possibility distribution π_A gives the certainty degree that a wall position takes the value (x,y) knowing the fuzzy set F_s.

From the possibility distribution π_A, the Possibility Π_A and Necessity N_A constitute two dual measures that are defined over the set□$P(U)$ of subsets of U:

$$\Pi_A, \mathrm{N}_A : P(U) \to [0,1]$$

$$\left.\begin{array}{l} \Pi_A(B) = \sup\{\pi_A(u);\ u \in B\} \\ \mathrm{N}_A(B) = \inf\{1 - \pi_A(u);\ u \notin B\} \end{array}\right\} \Rightarrow \mathrm{N}_A(B) = 1 - \Pi_A(\neg B)$$

Where A is a fuzzy subset of U and B as a crisp subset. In this manner, given the imprecise information "X is A", $\Pi_A(B)$ and $\mathrm{N}_A(B)$ measure the uncertainty of imprecise affirmations such as "X is B".

Finally, we provide a brief list of some of the Possibility Theory axioms and properties:

$$\Pi, \mathrm{N} : P(U) \to [0,1]$$

$$\mathrm{N}(A) = 1 - \Pi(\neg A)$$

$$\Pi(U) = 1, \quad \Pi(\varnothing) = 0, \quad \mathrm{N}(U) = 1, \quad \mathrm{N}(\varnothing) = 0,$$

$$\Pi(A \cup B) = max(\Pi(A), \Pi(B)), \quad \mathrm{N}(A \cap B) = min(\mathrm{N}(A), \mathrm{N}(B))$$

$$\Pi(A \cup \neg A) = 1, \quad \mathrm{N}(A \cap \neg A) = 0$$

$$\Pi(A) \geq \mathrm{N}(A), \quad \Pi(A) < 1 \Rightarrow \mathrm{N}(A) = 0, \quad \mathrm{N}(A) > 0 \Rightarrow \Pi(A) = 1$$

Ignorance is expressed by : $\Pi(A) = 1$ and $\mathrm{N}(A) = 0$

The Possibility Π and Necessity N values of singletons are 0.

7.2 Modelling the Certainty

7.2.1 Information Representation

The space being explored by the robots is discretized by means of a grid. Cells in the grid represent a small area of the real environment. The size of this area depends on the granularity that the host uses to represent the environment. Our system considers square cells whose side size is a parameter of the system (it is usually taken as 1 or 2 cm long). Grid cells contain two values: the degree of possibility and the degree of necessity of the presence of obstacles. Initially, all the cells have a possibility value Π of 1 and a necessity value N of 0. These initial values correspond to a situation of total ignorance and are intuitively interpreted as 'although it is completely possible that there is a wall, there is no certainty at all about it'.

As the host receives robots' partial maps, it modifies the possibility and necessity values of the global map grid. Value update is done in a way that depends on the presence, or absence, of obstacles. As we have already seen, partial maps contain robot trajectory segments together with information about which trajectory segments where covered while following a wall. If wall following occurs, then the partial map also specifies which side of the robot detected the wall and if the end of the segment corresponds to a singular point or not (i.e., if the wall ends). This additional information allows the host to compute the wall segment that corresponds to the trajectory segment.

The last section in the second chapter (see Sect. 2.4.4) indicates how localisation error rectangles are associated with the points that belong to both, robot trajectory segments and followed wall segments. These rectangles are used to determine the cells in the grid whose possibility and necessity values will be updated.

Although the host receives information specified in segments —in the continuous space—, it works with a discretized representation of the environment. Therefore, the host needs to discretize both wall and trajectory segments. As a result, discretized segments become a sequence of consecutive positions that correspond to grid cells. Rectangle errors are associated to each segment position so that possibility and necessity values are updated for every cell that results partially or totally covered by the rectangle area. Next Figure 7.1 a) depicts an example of a robot position centred in its rectangle area. Part b) corresponds to all the positions (and their associated rectangle errors) in a segment.

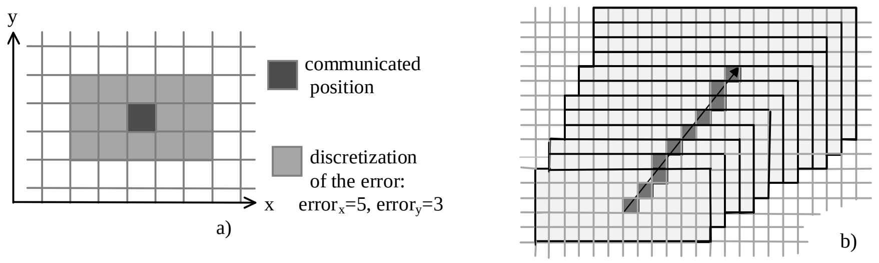


Figure 7.1: a) Grid representation of a position and its associated error; b) Consecutive positions of a discretized segment.

In the sequel, we introduce the fact that the area described by the consecutive rectangle errors associated with each position of these discretized segments determines which grid cells must be marked with occupancy degrees and which ones with free-space certainty degrees. We also give an intuition of how these certainty degrees are distributed along the grid in a pyramidal manner.

Walls and Obstacle Edges Representation

When representing a wall position (or an obstacle edge) and its associated error rectangle, the certainty degree is updated in every cell that results partially or totally covered by the rectangle area. Information about detected and followed walls updates the necessity value in the corresponding cells and represents the certainty about the presence of an obstacle in each position.

Necessity values decrease linearly with the magnitude of the error and remain positive (N(wall) =α> 0) in the cells inside the error rectangle but obtain the value 0, in what concerns this particular detection, at the cells outside the limits of the rectangle. These values have been established with the aim of reflecting that, having followed some wall or obstacle, the necessity that there is a wall cannot be longer zero but positive, since a positive value denotes some certainty about the occupancy of the space. However this occupancy certainty degree decreases when the distance to the central cell (of the error rectangle) increases. Figure 7.2 a) gives an idea of how these values are distributed. Notice that the possibility value is constantly equal to 1 in all the cells covered by the error rectangle. This is due to the axiom of Possibility theoryN(A)>0⇒Π(A)=1.

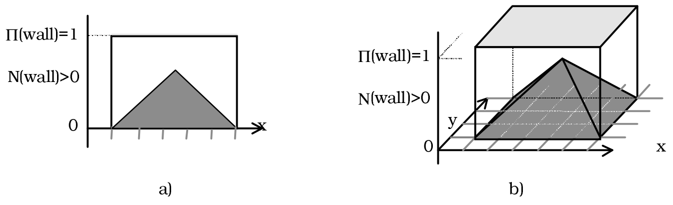


Figure 7.2: Π and N value assignment for a wall position. a) Projected to one dimension, and b) Two dimensions

Free Space Representation

Robot trajectory segments provide information about free space, meaning that the possibility of the existence of a wall has decreased with respect to the initial ignorance values.

In terms of possibility and necessity values, free space information corresponds to Π(¬wall)=1 and N(¬wall)>0. However, the global map grid only represents occupancy information (that is, Π(wall) and N(wall)). Fortunately, the possibility theory axiom N(A)=1-Π(¬A) allows us to transform

N(¬wall)>0 values into Π(wall)<1 as well as Π(¬wall)=1 values into N(wall)=0.

Possibility values of free-space positions Π(wall) increase inversely to the decrement of necessity values of N(¬wall). In the same way as necessity values of wall positions decrease proportionally to the distance to the central cell of the error rectangle, the possibility value of obstacles Π(wall) increases with the same proportion until it reaches the value 1 at the cells outside the limits of the error rectangle. (The possibility of existence of walls or obstacles outside the error rectangle around a trajectory in free space is 1). Obviously, we have N(wall) = 0 for all the cells covered by the error rectangle. The following Figure 7.3 depicts this inverted pyramid representation in the grid.

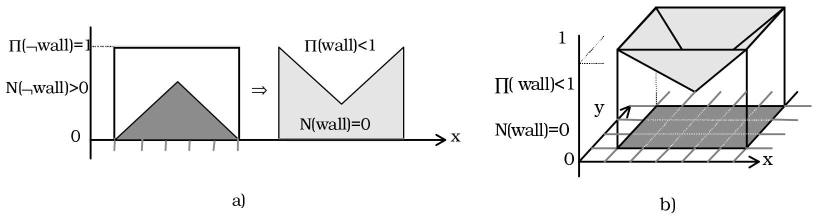


Figure 7.3: Representation of a free-space position: a) Transformation from Π(¬wall) and N(¬wall) into Π(wall) and N(wall). b) Representation of the obtained values Π(wall) and N(wall) in two dimensions.

7.2.2 Value Assignment

The height of the pyramids in the previous Figure 7.2 and Figure 7.3 is determined by the size of the error rectangle. The underlying idea is to establish a linear error-height relation such that, a null error implies the maximum allowed value of height (i.e. one), and an error equal to the maximum allowed error, K, implies a zero height since the information is no longer reliable. The maximum allowed error threshold, measured in grid units, assigns a limit to the error. K is established experimentally and is the same as the one that forces the robot to return from its exploration due to the irrelevancy of its later data.

Let us represent the error rectangle by the tuple $\{x_c, y_c, e_x, e_y\}$, where x_c and y_c are the co-ordinates of the central cell of the rectangle and e_x and e_y are, respectively, half the length of the base and half the length of the height of the error rectangle measured in number of grid cells. Following this, the height of the pyramid is given by:

$$height = 1 - \frac{max(e_x, e_y)}{K} \qquad (1)$$

And the necessity value of having a wall at cell (i, j) is given by:

$$N_{ij}(wall) = height * min\left(1 - \frac{|i - x_c|}{e_x}, 1 - \frac{|j - y_c|}{e_y}\right) \qquad (2)$$

The Π_{ij}(wall) values are obtained through 1-N_{ij}(¬wall) by using the same (1) and (2) equations.

Local Assignment

The same value assignment can be implemented by locall y computing height values for every grid cell that results partiall y or totally covered by the rectangle area. The computation of this height is done by means of a value propagation that starts at the central cell and spreads over all those cells laying within the pyramid base. Such a propagation passes four different values among cells: e_r, e_l, e_u, and e_d. These values contain the distance between the current position and each side of the error rectangle, i.e. right, left, up and down respecti vely. Since each value is passed from a cell towards its immediate neighbours, all values are unitaril y increased or decreased in each step of the propagation until they reach the zero value. Let $error_x$ be the length of the error rectangle base, and let $error_y$ be the rectangle height, then the error values are initiall y assigned at the central cell as follows:

$$e_r = e_l = \frac{error_x}{2}, \quad e_u = e_d = \frac{error_y}{2}$$

Then, the formulas that each cell uses to compute its height (that is, its N value) are the following:

$$N = min(N_x, N_y)$$

$$N_x = 1 - \frac{x}{err_x} + \frac{x - err_x}{K}, \quad \text{where: } x = \frac{|e_l - e_r|}{2}, \quad err_x = \frac{e_l + e_r}{2}$$

$$N_y = 1 - \frac{y}{err_y} + \frac{y - err_y}{K}, \quad \text{where: } y = \frac{|e_d - e_u|}{2}, \quad err_y = \frac{e_d + e_u}{2}$$

The next Figure 7.4 depicts an example of an error rectangle on the grid with $error_x$ = 8 and $error_y$ = 4. The propagation starts at the central cell, whose necessity value is the highest. Whenever a cell receives a null propagation value, its necessity value becomes zero, and since negative values cannot be propagated, the propagation finishes. These cells coincide with the edges of the error rectangle.

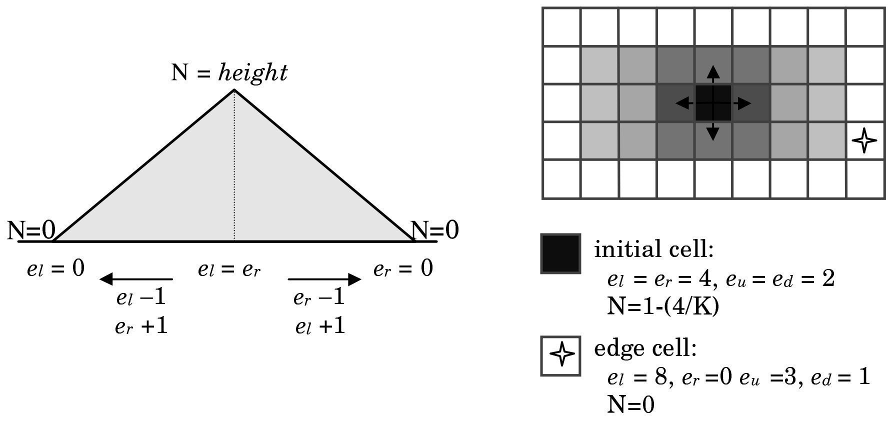


Figure 7.4: Local value propagation in order to compute necessity values N for each cell in the rectangle area.

With the aim of generating regular necessity pyramids, the propagation process has been implemented symmetrically. This forces us to solve the cases with even rectangle length with an ad-hoc solution: we markthe two central cells as initial. This choice is purely arbitrary, but it does not seem to be worse than any criteria that always takes the same semi-centred position.

The next Figure 7.5 shows several examples of pyramids generated by the host computer. A wall position generates a Necessity pyramid in red (1) whereas inverted possibility pyramids appear in blue (2 to 6). In both cases, darker colours mean a higher certainty degree: darker red in a cell stands for a higher necessity value of having a wall in this cell; a darker blue indicates that the possibility of having a wall is smaller (inverse to the necessity of free space). This figure contains several examples ofPossibility pyramids. They appear in an increasing error order. As we can observe in the figure, a large error rectangle associated to a position not only implies that a larger number of cells will be covered in the grid but also that they will receive smaller certainty values (i.e., lighter colours).

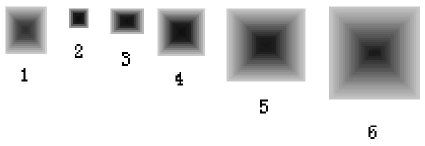


Figure 7.5: Example of the representation in the host of a Necessity pyramid (1) and several Possibility pyramids (2 to 6).

Finally, it is important to notice that these values of N>0 do not correspond to single points (i.e., singletons) for which Possibility Theory would assign by definition a zero value, but to discretization intervals over the environment grid, and consequently they do not have to be zero.

7.3 Global Map Generation: Certainty Grid Building

As we have said, the host generates the global map incrementally, by including the information of new income partial maps. The host treats partial maps sequentially, in the same order as they have been received. Since all partial maps are treated in exactly the same manner, this section details how a partial map is included into the global map. This process is repeated as many times as the host receives new partial maps. Each partial map includes the number of the robot it comes from. This is necessary because the host needs to know the characteristics of the robot that delivered the partial map in order to compute associated errors and detection distances. Nevertheless, this robot identification is not only necessary to treat partial map information but also to avoid processing more than once the information coming from one robot.

7.3.1 Partial Map Inclusion

The previous chapter explains how robots include their exploration information in partial maps. Basically, a partial map consists in a sequence of robot's trajectory segments. The host computer treats the information in the same order: for each new trajectory segment it reads, it includes the corresponding certainty values in the global map grid. The next algorithm in pseudo-code presents how the host adds each new partial map:

```
Add_new_partial_map_into_global_map( partial_map, maximum_allowed_error)
{ Repeat while( partial_map ≠ ∅ )
  {  segment = read_segment(partial_map)
     error = max_error(segment)
     If( error < maximum_allowed_error )
     {  d_seg = discretize (segment)
        If ( segment.wall ≠ 'no' )
        {  d_seg = orthogonalize(d_seg)
           d_wall=compute_wall_segment( d_seg )
           add_wall_and_trajectory_to_global_map( d_wall, d_seg )
        }
        Else
          add_trajectory_to_global_map(d_seg)
     }
     Else
        partial_map = ∅
  }
}
```

As the algorithm above shows, every trajectory segment is discretized and included into the global map grid. Nevertheless, trajectory segments that where covered by the robot while following a wall are treated separately. Before including them into the grid map, the host orthogonalizes them. They become vertical or horizontal whenever they are 'almost' collinear to one of the axis. This collinearity condition is measured in terms of the change rate between in the x the y co-ordinates of the segment. This is equivalent to the angle that the segment defines with one axis, but computationally simpler. In the 7.4 Results Section we will be able to observe the advantages of segment orthogonalisation and the effects of setting the 'almost' collinear condition to different angles. By now, we just point out that when this condition holds for a segment, the components with less variation become equal to their mean value. Clearly, oblique segments are not changed.

Wall following trajectory segments contain information that allows the host to compute the corresponding wall segment. Considering the robot displacement and the side of wall detection, the host generates a wall segment parallel to the trajectory segment. Both segments are separated by a distance that corresponds to the robot detection distance. (Section 1.1 already explained the error associated with the resulting wall segment).

Since the robot odometry error increases with the covered distance, segments appear in the partial map in an increasing error order. They are included into the global grid map only if their error is smaller than a maximum allowed error. Such a system parameter indicates a data relevance threshold so that beyond its value, no more segments are considered and the algorithm ends. Segments are discretized and each resulting point has an associated rectangle error. From the error accumulation, we can ensure that the size of the rectangles increases from the initial point of the segment until the final point. Therefore, it is enough to check if the rectangle error associated with the last point is larger than the maximum allowed error. And it is considered to be the case when one of the two rectangle dimensions is larger than this threshold.

The present section continues by explaining how trajectory and wall segments are added to the map grid using two distinct procedures. However, before this, it is worth to briefly describe the structure of the grid map in the system.

Map Grid Structure

A map grid is an object class that contains, as members, the dimensions and the origin of the grid together with a two dimensional dynamic array of grid elements. (The grid is built at execution time. However, we will not go into the detail of commenting its methods).

Grid elements correspond to the so called up to now grid cells. Each element has the following members:

- Necessity value. This value is a real number that belongs to the [0,1] interval. It represents the necessity degree that the cell is occupied.
- Possibility value. As the necessity value, it is a real number between 0 and 1. This value represents the occupancy possibility of a cell.
- Singular point. Its value is a label for the cell: '1' means that the cell corresponds (with the necessity degree contained at the cell) to a singular point. Otherwise, its value is '0'.
- Orientation. This value is again associated to wall positions only. Its value is '0' when the cell only contains information about trajectory positions. Otherwise, it can take three different values that distinguish among horizontal, vertical or oblique orientations: HOR, VERT, and OBL respectively.

According to Possibility Theory, possibility and necessity values in a cell must fulfil the axioms in the 1.1 Section. That is, for every cell:

- The Possibility value must be always larger than the Necessity value.
- A positive (non-zero) Necessity value implies a Possibility value equal to 1.
- A Possibility value smaller than 1 forces the Necessity value to be 0.

In the previous section we explained that we can represent free space information by assigning Possibility values smaller than 1 to the cells. We will further discuss in the rest of this section why we do not fulfil these restrictions in intermediate stages of the map building process. This allows us to combine different information in order to choose the most relevant one at the end of the process: when displaying the map.

7.3.2 Value combination

In Sect. 7.3.1 we have seen an algorithm that includes partial maps. This algorithm discretizes each trajectory segment and includes the information it contains in the global grid map. In this subsection we see how this is done.

Including Trajectory Information in the Global Map

The discretization of a trajectory segment results in a sequence of grid positions. Each of these positions has an associated rectangle and represents free-space. Section 1.1 explains how a free-space position and its error can be represented in a grid by means of an inverted pyramid of possibility values in the corresponding cells.

A segment is defined by its initial and final points together with their associated errors. The discretization process is based on the Digital Differential Analyser (Hearn 88), which is an incremental algorithm for drawing lines in digital devices. Utilising this algorithm, cells in the grid are chosen in the same way than pixels in a screen. Moreover, in the same way that the first and last position in a trajectory allows to compute all intermediate cell positions, their associated rectangle errors can similarly be computed and associated with each new position.

In this manner, the function that includes trajectory information into the global grid map performs two main steps. First, it discretizes the corresponding trajectory segment. And second, for each resulting trajectory position, it computes its error and includes the corresponding possibility pyramid into the grid. In the previous subsection we have seen how cell necessity and possibility values are propagated from a central cell towards its surrounding cells. As a consequence of this discretization, central cells are consecutive and therefore, value propagation is done over cells that may already have been updated by a previous propagation. In other words, we are propagating overlapping pyramids, and this implies that new values must be the result of a combination between old and new pyramids.

This value combination must guarantee that the addition of new information always reduces the ignorance at any given cell. Taking into account that ignorance is expressed with the maximum value of occupancy possibility ($\Pi = 1$) and the minimum value of occupancy necessity ($N = 0$), and considering also that free space information is translated into possibility values smaller than 1, we can conclude that to combine possibility values with the *minimum* operator guarantees this ignorance reduction. (Note that this *min* operator has been described in Sect. 3.1.1 as a fuzzy set t-norm operator).

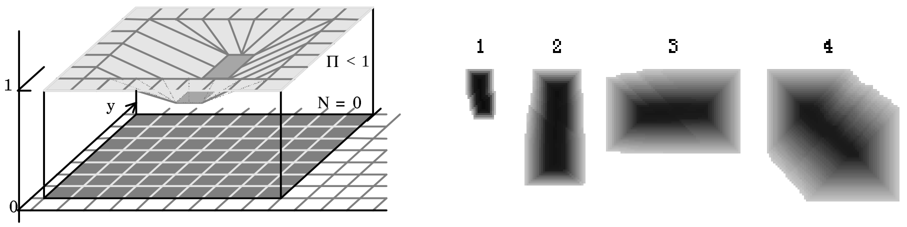


Figure 7.6: Free-space representation. (Left): As a combination of consecutive inverted possibility pyramids. (Right): Four different trajectories included in the global map.

The previous Figure 7.6 depicts on the left the idea of combining consecutive possibility pyramids using the *min* operator. Figure 7.6 (right) contains trajectory positions that have been included in the global map. The represented information comes from four different trajectory segments. The first one has a reduced error and, since the error increases in a bottom-up manner, it implies that the robot moves upwards. As the previous example, the second example corresponds to vertical robot displacement. Nevertheless, its error is less moderated and the robot moves downwards. The remaining to examples come from a horizontal and an oblique robot trajectories respectively. Both present a relatively large error.

Including Wall Following Information in the Global Map

When a trajectory segment has been covered while following a wall, first thing to do consists in orthogonalising it. We have already commented that a segment becomes completely vertical or horizontal only if it already is almost orthogonal. Afterwards, the information of two segments is included in the global map: the trajectory segment and the corresponding wall segment. The trajectory segment is included as the previous subsection details. The wall segment representation implies a combination of consecutive necessity pyramids. Wall positions propagate pyramids with positive values increasing from 0 at the rectangle error edges. We combine the necessity values of consecutive pyramids with the *maximum* operator (see Figure 7.7). This operator corresponds to the fuzzy t-conorm and has been chosen in order to maintain occupancy certainty values as high as possible.

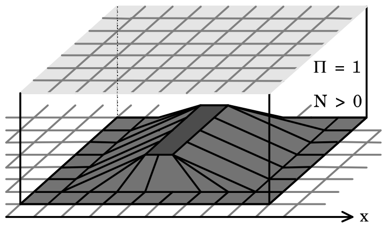


Figure 7.7: *Max*-combination of consecutive necessity pyramids.

Both, wall and trajectory segments have the same length, and therefore, they can be discretized and included in the map simultaneously. However, wall information is included into an auxiliary grid map whereas trajectory information is directly propagated on the global map grid. This auxiliary grid map is created in order to fit the wall information coming from the current wall segment. This allows to combine necessity values coming from the same wall following without mixing with previous values in the global grid.

Once all the wall positions have been included into the auxiliary grid, it is transferred into the global grid map at the corresponding position. Unlike trajectory information, wall information cannot be directly merged with the necessity values in the global map. This is because two distinct operations are involved in each combination. First, the *maximum* operator is applied for necessity propagation inside the auxiliary map. And second, the wall combination operator (between necessity values in an auxiliary grid cell and its corresponding cell in the global map) is the Probabilistic Sum $S(x, y) = x + y - x \cdot y$. This operation has been chosen with the aim of reinforcing the wall occupancy certainty in those cells in which two or more different sources of information coincide. By different sources we mean that this piece of wall has been followed more than once. Once the information of a wall following has been included into the global map, it is not possible to distinguish who has followed a wall. Therefore, when combining previous wall information with the information in the auxiliary grid, we do not distinguish whether the wall has been followed by the same robot or by a different one, we always consider this information to be independent. Finally, note that the probabilistic sum operation does not change a necessity value N>0 when it is added into a cell having a zero necessity value.

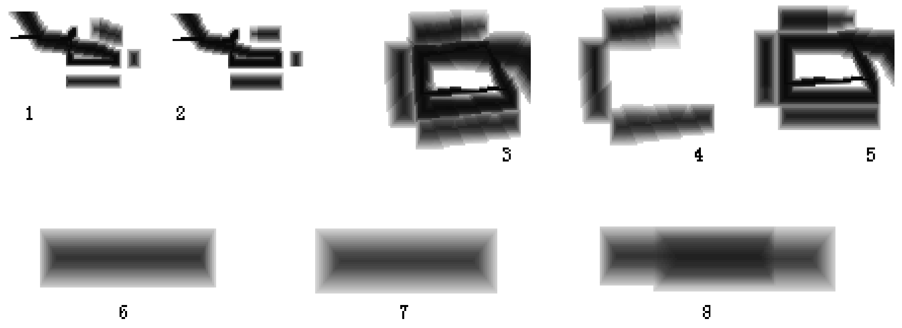


Figure 7.8: Several examples of wall representation. Walls and trajectories without being orthogonalized (1, 3, and 4). The corresponding orthogonal walls and trajectories (2 and 5). Wall reinforcement: (8) results from the combination of (6) and (7). The upper and left walls in 3, 4, and 5 correspond as well to the combination of two walls. Singular points appear in green at wall extremes.

At the beginning of this section, we saw the pseudo-code of the function Add_new_partial_map_into_global_map(). In order to include wall following information into the global map, this function calls another function whose name is add_wall_and_trajectory_to_global_map. Although we have already commented all the steps that this function implements, we present its pseudo-code with the aim of summarising them. Figure 7.8 contains examples of the way in which the host represents and combines wall following information. The images inside the figure illustrate the function steps.

```
add_wall_and_trajectory_to_global_map( d_wall, d_seg )
{  orthogonalize( d_seg, d_wall )
   dim = compute_wall_dimensions( d_wall )
   orient = compute_wall_orientation( d_wall )
   aux_map = create_auxiliary_map( dim )
   Repeat ( for each discrete position p in d_seg )
   {   propagate_possibility_values_in_global_map( p , min )
       wall_p = take_the_corresponding_wall_position( p )
       sing_pt = check_if_the_wall_position_is_a_singular_point( wall_p )
       propagate_necessity_values_in_auxiliary_map( aux_map, wall_p, orient, sing_pt, max )
   }
   add_auxiliary_map_into_global_map ( aux_map, probabilistic_sum)
}
```

Auxiliary Maps have the same structure as the global map. The Map Grid Structure subsection (page 179) specifies that each cell has two associated labels: singular point and orientation information. These labels take values only if the cell contains wall information. That is, if the cell has a positive value in its necessity field. In this manner, when propagating the wall information that a wall segment implies, the orientation label is simultaneously assigned. Regarding the singular point label, it can be only assigned in the propagation of the last position if the robot labelled the corresponding segment in the partial map as having a singular point. As the first five images in the previous Figure 7.8 shows, singular points describe the same necessity values distribution than the rest of wall positions in a wall, but they appear in green.

7.3.3 Local propagation

The method used to update wall information in the global map (that is, the combination operation we have just seen) takes advantage of the fact that wall information is given as segments that come from a single wall following. This allows to reinforce overlapping —but independent— wall segments. Nevertheless, this presents the disadvantage of forcing the use of an auxiliary grid map. This implies that the updating process can not be considered as being purely local: it is local at the segment level but not at the cell level. If we want to remain purely local, the computation of the necessity value of a cell must be done considering only those cells surrounding that cell.

We propose here an alternative method to include wall information. This method is purely local. Nevertheless, with the information stored in the cells it is not possible to distinguish whether two necessity values are independent or not. Or in other words, the algorithm cannot know if new propagated values that arrive to a given cell come from the same wall following or not. Therefore, there is no way of discerning when two necessity values should be combined by using the maximum operator or applying the probabilistic sum. This restriction forces local propagation to use only one operation to combine necessity values. We have chosen the most conservative one: the maximum operator. Being conservative reflects the fact that we prefer not to reinforce independent wall information than to reinforce non-independent wall information.

We compute the height —i.e., necessity value— of the central cell of the error rectangle, at position (i,j), with the same formula (1) introduced previously:

$$n_{i,j}^{t}(wall \mid \langle i,j,e_x,e_y \rangle) = 1 - \frac{max(e_x, e_y)}{K} \qquad (3)$$

Next, we need to define the propagation of this measure from any cell (n,m) within the error rectangle to its four neighbours. The propagation follows the following inequalities:

$$n^t_{n-1,m}(wall \mid \langle i,j,e_x,e_y\rangle) \geq n^t_{n,m}(wall)\cdot max(0, 1-\frac{|(n-1)-i|}{e_x}) \quad (4)$$

$$n^t_{n+1,m}(wall \mid \langle i,j,e_x,e_y\rangle) \geq n^t_{n,m}(wall)\cdot max(0, 1-\frac{|(n+1)-i|}{e_x}) \quad (5)$$

$$n^t_{n,m-1}(wall \mid \langle i,j,e_x,e_y\rangle) \geq n^t_{n,m}(wall)\cdot max(0, 1-\frac{|(m-1)-j|}{e_y}) \quad (6)$$

$$n^t_{n,m+1}(wall \mid \langle i,j,e_x,e_y\rangle) \geq n^t_{n,m}(wall)\cdot max(0, 1-\frac{|(m+1)-j|}{e_y} \quad (7)$$

We use inequalities because we want to keep the maximum of the different propagated values. For instance, if a propagation is made from cell (*n, m*) to cell (*n*-1, *m*), then the propagated value that (*n*-1, *m*) receives is smaller than the necessity at (*n, m*). Although the inequalities allow to compute afterwards a new propagation back from (*n*-1,*m*) to (*n,m*), the '≥' symbol prevent (*n*, *m*) to change its necessity value because the propagated value will be smaller than the one at (*n,m*). The propagation finishes when the values get stable in all cells.

Finally, the N measure is updated as follows:

$$N^t_{n,m}(wall \mid \langle i,j,e_x,e_y\rangle) = max(N^{t-1}_{n,m},\, n^t_{n,m}(wall \mid \langle i,j,e_x,e_y\rangle)) \quad (8)$$

Once again, free-space can be computed with the previous formulas (4 to 8) from $N^t_{n,m}(\neg wall \mid \langle i,j,e_x,e_y\rangle)$, which is translated into possibility values $\Pi^t_{n,m}(wall \mid \langle i,j,e_x,e_y\rangle)$.

The next Figure 7.9 depicts the results of combining wall information with the *maximum* operator. This figure includes three different combinations of pairs of walls. With the aim of illustrating the difference between using reinforcement or not, these walls are the same ones that appeared in Figure 7.8.

Figure 7.9: Local necessity combination using the max operation. The walls are the same ones as in Figure 7.8.

7.4 Possibilistic, Probabilistic, and Evidential Models

Both possibilistic and probabilistic models are particular cases of the evidential model (Shafer 76, Smets 88). In the evidential model, a function m assigns a mass to each subset of the frame of discernment Ω in the following way:

$$m : P(\Omega) \to [0,1]$$
$$m(\varnothing) = 0$$
$$\sum_{A \subset \Omega} m(A) = 1$$

A mass assignment m generates two dual measures of Belief and Plausibility defined as:

$$Bel(A) = \sum_{B \subset A} m(B)$$
$$Pl(A) = \sum_{B \cap A \neq \varnothing} m(B)$$

So that they verify:

$$Bel(A) \leq Pl(A)$$
$$Pl(A) = 1 - Bel(\bar{A})$$

In general, the masses can be distributed in any manner. The possibilistic model corresponds to the case where these masses are nested. Then, the belief and plausibility measures correspond to necessity and possibility measures respectively. On the other hand, a mass assignment generates a probability if the focal elements (that is, these subsets having positive masses) do not intersect.

In our case, the frame of discernment is Ω = {wall, ¬wall}, and our mass functions of interest are simple support masses of the following type:

$$m: P(\Omega) \to [0, 1]\,,$$
$$m\,(\varnothing) = 0,$$
$$m\,(\{\text{wall}\}) = \alpha,$$
$$m\,(\{\neg\text{wall}\}) = 0,$$
$$m\,(\Omega) = 1-\alpha.$$

Therefore, we obtain:

$$Bel(wall) = \sum_{A \subseteq \{wall\}} m(A) = m(\{wall\}) + m(\varnothing) = \alpha$$

$$Pl(wall) = \sum_{A \cap \{wall\} \neq 0} m(A) = m(\{wall\}) + m(\Omega) = 1$$

Ignorance is represented by assigning the total mass to the entire frame (m(Ω) = 1) without assigning any mass to its subsets. On the contrary, total security of having wall would be represented by assigning the entire mass to the wall set (m({*wall*}) = 1) exclusively.

Our *m* function assigns positive masses to the wall set *m*({*wall*}) = α and the complete frame m(Ω) = 1 – α, leaving the non-wall set *m*({¬*wall*}) with a null assignment. In this manner, we are only considering nested sets ({*wall*} ⊂ Ω), and therefore, we can consider Belief(wall) and Plausibility(wall) as being Necessity and Possibility measures respectively. In this manner, we have N(*wall*) = α > 0 and Π(*wall*) = 1.

Having a evidence of wall, we can intuitively interpret this mass assignment in the following way: since the evidence is not completely sure, we assign a positive mass α to the wall set, a null mass to non-wall (we do not have evidence for non-wall), and leave the rest (1 – α) to the frame as a measure of incompleteness of our information.

If, instead of assigning m(Ω) = 1 – α we would assign this mass to the complement of the wall set (that is, *m*({¬*wall*}) = 1 – α), then, these measures would correspond to a probabilistic model. This is because, under these circumstances, we are considering {*wall*} and {¬*wall*}, which do not intersect. In the Section 1.1 we already commented that the reason of choosing possibility/necessity techniques instead of probability is our need of an initial assignment of values representing ignorance. Possibility theory allows a clear representation of ignorance but probability does not, and forces the initial assignment to be: *m*({*wall*}) = *m*({¬*wall*}) = 0.5.

Following the interpretation of the Possibility/Necessity assignments as Belief/Plausibility values, we can justify now the use of the combination rules. On one hand, we have already seen that we apply the probabilistic sum when combining independent wall detections in the same cell, and this operation is nothing but Dempster's rule for simple support masses:

$$m_1, m_2 : P(\Omega) \to [0,1]$$

$$m_{1,2}(\varnothing) = 0$$

$$m_{1,2}(C) = \frac{\sum_{A \cap B = C} m_1(A) \cdot m_2(B)}{\sum_{A \cap B \neq \varnothing} m_1(A) \cdot m_2(B)}$$

In our case, two evidences assign the following masses:

$$m_1(\{wall\}) = \alpha_1 \quad m_2(\{wall\}) = \alpha_2$$
$$m_1(\Omega) = 1 - \alpha_1 \quad m_2(\Omega) = 1 - \alpha_2$$
$$m_1(\{\neg wall\}) = m_2(\{\neg wall\}) = 0$$
$$m_1(\emptyset) = m_2(\emptyset) = 0$$

And the resulting mass, using the Dempster-Shafer combination rule is:

$$m_{1,2}(\{wall\}) = \frac{\sum\limits_{A\cap B=\{wall\}} m_1(A)\cdot m_2(B)}{\sum\limits_{A\cap B\neq\emptyset} m_1(A)\cdot m_2(B)} =$$
$$= \frac{m_1(\Omega)\cdot m_2(\{wall\}) + m_1(\{wall\})\cdot m_2(\Omega) + m_1(\{wall\})\cdot m_2(\{wall\})}{m_1(\Omega)\cdot m_2(\Omega) + m_1(\Omega)\cdot m_2(\{w\}) + m_1(\{w\})\cdot m_2(\Omega) + m_1(\{w\})\cdot m_2(\{w\})} =$$
$$= \frac{(1-\alpha_1)\cdot\alpha_2 + \alpha_1\cdot(1-\alpha_2) + \alpha_1\cdot\alpha_2}{1} = \alpha_1 + \alpha_2 - \alpha_1\cdot\alpha_2$$
$$m_{1,2}(\{\neg wall\}) = \frac{\sum\limits_{A\cap B=\{\neg wall\}} m_1(A)\cdot m_2(B)}{\sum\limits_{A\cap B\neq\emptyset} m_1(A)\cdot m_2(B)} = 0$$
$$m_{1,2}(\Omega) = (1-\alpha_1)\cdot(1-\alpha_2) = 1-(\alpha_1+\alpha_2-\alpha_1\cdot\alpha_2)$$
$$m_{1,2}(\emptyset) = 0$$

On the other hand, we also combine values coming from a single wall following, and since we are considering non-independent evidences, Dempster's rule is not suitable for evidence combination. Instead, we have used a max-combination, a more cautious operation, still under the evidence theory framework. Indeed, max combination is in accordance with the so-called 'combination of compatible Belief functions' (Chateauneuf 94) that makes sense when interpreting Bel/Pl values as bounds of the probability measures consistent with them. Namely, let

$F_i = \{P \mid Bel_i \le P \le Pl_i\}$ i=1,2

be the family of such probabilities. Then, their natural combination can be taken as the intersection:

$F_{1,2} = F_1 \cap F_2 = \{P \mid \max(Bel_i, Bel_2) \le P \le \min(Pl_i, Pl_2)\}$

In general,

$$Bel^P = \inf_{P\in F_1\cap F_2} P \quad \text{and} \quad Pl^P = \sup_{P\in F_1\cap F_2} P$$

are not a pair of Belief and Plausibility measures (Chateauneuf 94). However, in our particular case, this combination leads to a proper belief function. Indeed, the function Bel^P is defined as:

$$Bel^P_{1,2}(wall) = \inf_{P \in F_1 \cap F_2} P(wall) = max(Bel_1(wall), Bel_2(wall)) = max(\alpha_1, \alpha_2)$$

$$Bel^P_{1,2}(\emptyset) = m(\{\neg wall\}) = 0$$

$$Bel^P_{1,2}(\Omega) = 1$$

a belief function whose corresponding mass assignments are:

$m(\{wall\}) = \max(\alpha_1, \alpha_2)$,
$m(\emptyset) = m(\{\neg wall\}) = 0$,
$m(\Omega) = 1-\max(\alpha_1, \alpha_2)$

Moreover, in this particular case, this max-combination is also in accordance with a new combination operation proposed in (Torra 95).

7.5 Results

Section 1.1 presented a complete example of robot exploration. That example included the partial map generated by a robot exploring an environment. Therefore, it seems reasonable to start this results section continuing that example. The following Figure 7.10 illustrates how the computer host uses the information contained in the robot's partial map to generate a map of the environment.

Figure 7.10 a) contains the same trajectory that we already saw in Figure 6.15. From this trajectory and the partial map, we could see that the robot followed a total of 7 portions of walls (which means a 12.8% of wall coverage). Figure 7.10 b) presents the global map obtained from including the partial map information into a grid initialised with the ignorance values. For each cell, ignorance is represented by null necessity occupancy values and possibility occupancy values equal to 1. The host displays this specific combination of values in white. Due to the addition of a partial map, some cells in the grid change their initial values into positive necessity values or possibility values smaller than 1. The host represent these cells in red and blue respectively. The lighter their colour, the closer to ignorance they are. The certainty corresponding to free-space is proportional to the darkness of blue. In this manner, we can interpret blue areas as free-space areas and distinguish the robot trajectory in the map by following consecutive blue cells. On the contrary, we can think of red areas as being occupied by walls or obstacles whose side have been followed by the robot. Again, the darker the red, the higher the necessity value.

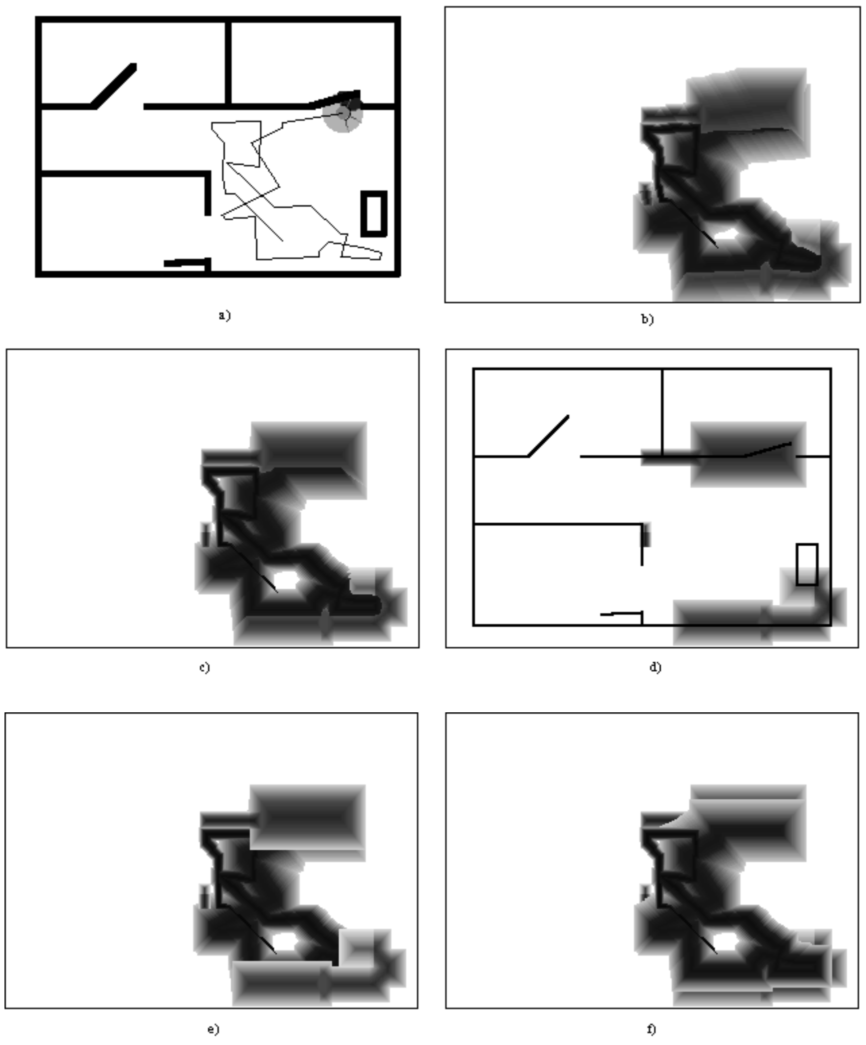


Figure 7.10. Map generation from the partial map information in the example of Sect. 6.4.1. a) Robot exploration. b) Non-orthogonalized map. c) Orthogonal map. d) Wall representation (the real map representation has been superposed for a better comparison of the results). e) An alternative manner of displaying the map: wall information has higher priority than trajectory information. f) The opposite display policy: possibility degrees over necessity degrees.

At the beginning, when including the information coming from a wall following segment, the detection distance is larger than the size of the accumulated rectangle error. Therefore, the cells receiving positive necessity values do not coincide with the ones that acquire possibility values smaller than 1. This implies that, on the one hand, the former cells do not vary their possibility values: they are equal to 1. And, on the other hand, cells with possibility values smaller than one still contain their necessity values equal to 0. Consequently, this assignment fulfil Possibility Theory axioms, which stand that:

$\Pi(A) \geq N(A)$, $\Pi(A) < 1 \Rightarrow N(A) = 0$ and $N(A) > 0 \Rightarrow \Pi(A) = 1$.

However, as the robot explores further, the accumulated error increases, and the occupancy and free-space information overlap. This means that the map generation algorithm will try to assign both positive necessity values (N_{ij}) and possibility values (Π_{ij}) smaller than 1 into single cells at (i, j) positions. As a solution, the algorithm can opt for the information with higher certainty value and apply the following assignment:

```
Necessity propagation(N)
{  If($\Pi_{ij}$ = 1) combine N with $N_{ij}$: $N_{ij} = N_{ij} + N - (N_{ij} * N)$
   Else
      If( $N \geq 1 - \Pi_{ij}$ ) assign $N_{ij} = N$ and force $\Pi_{ij} = 1$
}
Possibility propagation($\Pi$)
{  If( $N_{ij}$ = 0) combine $\Pi$ with $\Pi_{ij}$: $\Pi_{ij} = max(\Pi_{ij}, \Pi)$
   Else
      If( $1 - \Pi \geq N_{ij}$ ) assign $\Pi_{ij} = \Pi$ and force $N_{ij} = 0$
}
```

However, this assignment is order dependent. For example, if it is the case that a cell receives the following contradictory information: N=0.2, Π=0.7 and N=0.15. The resulting values are different depending on the order they are received. In case they arrive in the written order, the cell will end with $N_{ij} = 0$ and $\Pi_{ij} = 0.7$. This is due to that, initially, since $N_{ij} = 0$ and $\Pi_{ij} = 1$, the necessity value becomes $N_{ij} = 0.2$. Afterwards, for a new possibility value equal to 0.7, we have $1-0.7 = 0.3 > 0.2$ and, therefore, the cell values change into $N_{ij} = 0$ and $\Pi_{ij} = 0.7$. Finally, the last necessity value is 0.15 and therefore, it cannot change a 0.7 possibility value.

On the contrary, if the necessity values would have been combined before the arrival of the possibility value, the result would be different. This is due to the fact that two necessity values of 0.2 and 0.15 yield to a combined necessity value of 0.32. And this value cannot be varied by a 0.7 possibility value.

Instead of this order dependent solution we propose an order independent approach. In our approach we simply combine the received value with its equivalent at the cell:

```
Necessity propagation(N)
{ combine N with $N_{ij}$: $N_{ij} = N_{ij} + N - (N_{ij} * N)$
}
```

```
Possibility propagation($\Pi$)
{ combine $\Pi$ with $\Pi_{ij}$: $\Pi_{ij} = max(\Pi_{ij}, \Pi)$
}
```

And display the one with higher certainty:

```
Display_cell_value()
{  If($1-\Pi_{ij} > N_{ij}$) draw $\Pi_{ij}$
   Else
     If($1-\Pi_{ij} < N_{ij}$) draw $N_{ij}$
}
```

Although the internal representation of the map does not fulfil the Possibility Theory axioms, the host computer displays the resulting map satisfying the restrictions they imply. In fact, it allows the user to choose among four different preference criteria:

- display only walls (cells with $N_{ij} > 0$)
- walls and trajectories: blue if $1-\Pi_{ij} > N_{ij}$ and red if $1-\Pi_{ij} < N_{ij}$
- walls over trajectories: red if $N_{ij} > 0$, otherwise, blue if $\Pi_{ij} < 1$
- trajectories over walls: blue if $\Pi_{ij} < 1$, otherwise, red if $N_{ij} > 0$

In Figure 7.10 b) and c) the displayed cell values correspond to the ones with higher certainty value, whilst the map in d) exclusively displays positive wall necessity values. Note that c) is the orthogonalized representation of b). The third and fourth remaining criteria have been applied when displaying the map at the e) and f) images respectively.

The fact of assigning incompatible possibility and necessity values to cells is just an implementation issue. Instead of representing exclusively occupancy information, we could assign four different values for each cell: the possibility and necessity degree of being occupied together with the possibility and necessity degree of representing free-space. This would contain the same information. The only disadvantage is that it requires allocating twice the memory space for its representation.

7.5.1 Orthogonalization Results

Before considering other map generation examples, there is an additional aspect of the map representation that we want to comment: the orthogonalization of wall following segments. This is done by a simple function that plays an important role in map generation process. Firstly, it distributes the error uniformly by assigning to the positions at the extremes of the segment their mean error rectangles. And secondly, the fact of substituting less variant co-ordinates by their mean has two advantages. On the one hand, since wall following segments will become parallel, it favours the combination of independent representations. And, on the other hand, orthogonalized segments are closer to real maps. And this is not only because of our assumption of mainly orthogonal environments but because of robots' wall following performance. Often, a robot starts following a wall at a distance different from the distance it is when leaving the wall. And it is specially the case if a robot detects a singular point, because the robot approaches the wall more than usual (see Sect. 6.2.3). When computing the wall segment, the host does not have enough information to discern among different distances of wall following, and therefore, it seems reasonable to assign the wall co-ordinate mean. In the previous Figure 7.10 d) the real map representation appears superposed to the obtained map. Figure 7.11 a) below depicts the walls without being orthogonalized.

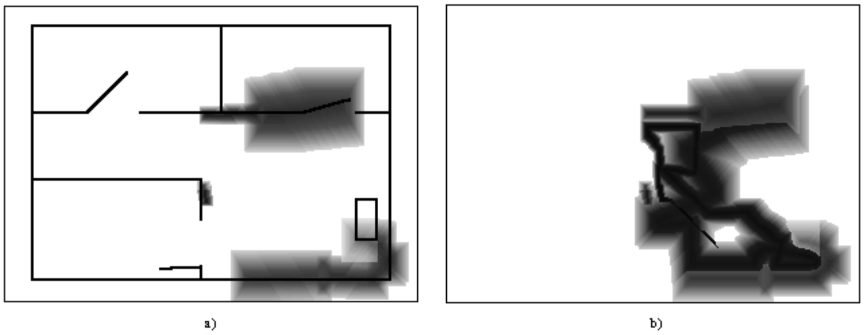


Figure 7.11: a) Non-orthogonal map representation. b) Orthogonal map representation using a threshold of 5.7°

By comparing with the real environment both maps at Figure 7.10 d) and Figure 7.11 a), we can conclude that most of the times, it is convenient to orthogonalize segments corresponding to wall followings. However, we can observe that the non-orthogonal information concerning the upper-right door is closer to the real environment than the orthogonal one. The decision of orthogonalization is made on the basis of the angle that the segment forms with the axis. In the host implementation, all segments determining an angle smaller than 26.5 degrees are orthogonalized. The following Table 1.1 lists the communicated trajectory segments and the angles they form with their closer axis (note that wall segments are computed to be always parallel to trajectory segments). None of these segments form an angle larger than 26.5° and therefore, all of them have been orthogonalized.

order	**initial position**	**final position**	**degrees**
1	(158, 75)	(156, 88)	8.7°
2	(148, 128)	(184, 129)	1.6°

3	(243, 28)	(272, 26)	3.9°
4	(272, 26)	(273, 31)	11.3°
5	(273, 31)	(259, 35)	15.9°
6	(227, 28)	(181,26)	2.5°
7	(200, 128)	(245,135)	8.8°

Table 7.1: Angles between wall following segments and their closer co-ordinate axis.

This angle threshold value is equivalent to a simple condition that checks if the length of the projection in one axis is twice the length of the projection in the other axis. This condition is implemented by a function that computes the orientation for each segment. The pseudo-code of the function is as follows:

```
Orientation( initial_point, final_point )
{  x_inc = |final_point.x-initial_point.x|
   y_inc = |final_point.y-initial_point.y|
   If( x_inc = 0 Or y_inc/x_inc ≥ 2 )
      orientation = Vertical
   Else
      If( y_inc = 0 Or x_inc/y_inc ≥ 2 )
         orientation = Horizontal
         Else  orientation = Oblique
  return ( orientation )
}
```

In this manner, if the function assigns a 'Vertical' or a 'Horizontal' label to a segment that is not completely horizontal, it must be orthogonalized. Otherwise it is considered as being 'Oblique'.

Obviously, this threshold can be changed. Figure 7.11 b) presents an example whose orthogonalization threshold has been narrowed to 5.7 degrees (a proportion of 10 instead of 2). Consequently, only 3 of the wall following segments have been orthogonalized. They correspond to the 2nd, 3rd,and 6th in the previous table. (In the figure they respectively are: the upper-left wall and trajectory segments, the bottom-right horizontal segments, and the bottom-right vertical segments).

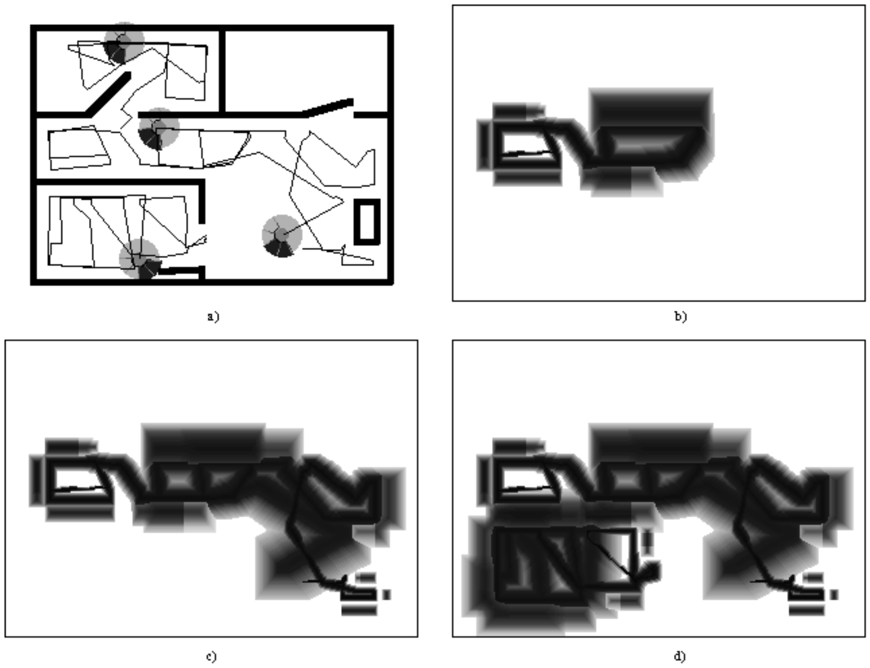
7.5.2 Global Map Generation Results

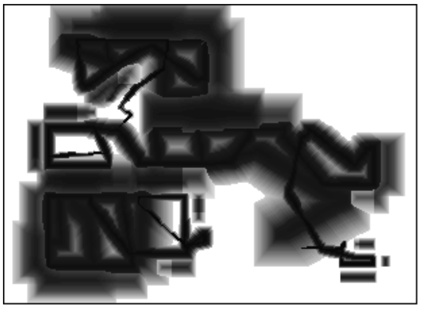
We have used the simulator described in Chapter 5 to simulate the exploration of four robots in the previous environment. All the robots have the same default error characteristics. They are internally numbered following the order in which they have been introduced into the environment. The four robots have been added in different initial positions and different probabilities have been assigned to each of them. The following Table 7.2 lists these characteristics. (The environment characteristics and the default robot features can be respectively found in Sections 5.2.2 and 5.3.1).

#robot	initial position	turning probability	l/r turning prob.
1	(75, 110)	20	65
2	(220, 40)	25	35
3	(135, 40)	35	55
4	(85, 130)	20	50

Table 7.2: Robot initialisation parameters.

The next Figure 7.12 a) contains a snapshot of the four robot's trajectories after having finished their exploration.




e)

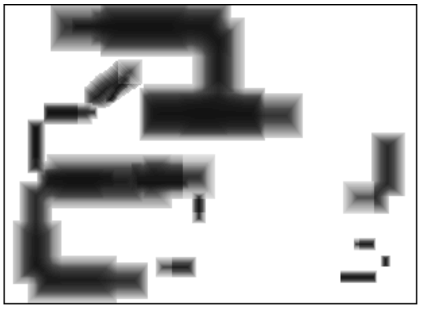

f)

Figure 7.12: a) Exploration of four robots. From b) to f) incremental generation of the global map based on the information provided by the four robots.

Robots' partial maps are communicated sequentially to the host computer. In the simulation, they are stored in files named after the environment and with the number of the robot at the end. Initially, the host processes the first partial map and includes its information into the global map representation. Figure 7.12 b) shows the current global map. Afterwards, the host reads the second partial map file and obtains the map shown in c). Image d) of this Figure 7.12 contains the global map including the partial maps coming from the first three robots. The final global map appears in e), and f) presents the wall information.

We have already presented results by other authors that state that a homogeneous distribution of initial robot positions in the environment increases environmental coverage when exploring randomly. Therefore, we are not going to discuss here robot initial positions but their wall coverage. The next Table 7.3 describes the information that each new partial map provides to the host, and how does this information affects the global map representation. The first six columns indicate partial map characteristics, whereas last three columns contain global map information. The first column determines the partial map order. The second column contains the number of trajectory segments in the partial map, which entail the total covered distance in the third column. From this total number of segments, only the ones at the fourth column correspond to segments that were covered while following a wall. During wall following, the robots detect a reduced number of singular points (i.e., wall ends); they are listed in the fifth column. Last partial map column contains the distance that the robot —that generated the partial map— covered while following walls. Distance units depend on the environment granularity, for this case, each unit corresponds to 2 cm. Regarding global map information, the values at their columns are accumulative. Each partial map information entails an increment in the number of features represented in the global map. First column presents the total distance (not area) of wall representation that coincides with real walls in the environment. When a wall that has been previously followed from one side is afterwards followed along the other side, the length of the represented walls does not increase. The second column lists the total number of represented singular points. And finally, last column gives the percentage of wall coverage.

Partial map information						**Global map**		
#	#t. seg.	total dist.	#w. seg.	#s.pt.	wall dist	wall dist	#s.pt.	coverage
1	36	621.3	9	4	362.2	242.8	2	16.5%
2	45	650.1	6	2	140.7	444.2	4	30.2%
3	45	758.7	10	2	363.6	672.2	6	45.8%
4	37	699.7	9	2	419.9	869	7	59.1%

Table 7.3: Global map analysis (distance units equal the environment granularity = 2 cm).

As we can observe from the values in Table 7.3, the increment in the wall length of the global map does not strictly depend on the wall following distance of the partial map, but only on the length of those portions of walls that had not been followed before. Nevertheless, although new wall information is always useful to increment wall coverage, the error associated to new information should also be taken into account. If the environment is relatively cluttered, wall information can close free-space areas. For example, we notice in Figure 7.12 f) that the free space between the obstacle and the wall appears to be closed when the error becomes too large. Obviously, the 10 units (that equals to 20 cm) that correspond to real free-space have not being added to the wall distance value. For this particular case, since this free-space cannot be reached by a robot because of its reduced size, it does not represent a significant problem. However, there are other areas, such as the upper left room, that have too many cells with positive occupancy necessity values. This means that the maximum allowed error was still too large. The host computer executes an application that is user driven. The user can decide when to include the next partial map, how the global map will be displayed, as well as the maximum error associated with information that will be allowed. Therefore, the partial map can include information that will not be added into the global map (that will happen if its associated error is larger than the error threshold given by the user).

#	**#t. seg.**	**ini.→ final co-ordinates**	**error**	**s.pt**	**wall**
1	5	(32, 108) → (32, 127)	4.75	no	left
2	6	(32,127) → (65, 132)	8	yes	left
3	10	(77, 103) → (33, 99)	12.36	no	left

4	11	(33, 101) → (31, 128)	11.01	no	left
5	12	(32, 128) → (59, 129)	13.6	yes	left
6	19	(99, 103) → (143, 100)	20.9	yes	right
7	24	(180, 127) → (111, 130)	29.6	no	right
8	29	(113, 103) → (143, 100)	30.03	yes	right
9	36	(179, 127) → (114, 130)	38.8	no	right

Table 7.4: Wall following trajectory segments from the partial map of the first robot.

Table 7.4 lists the trajectory segments in the first partial map from which walls have been computed. Each segment is identified by means of its order in the partial map (first column) and its initial and final co-ordinates before being orthogonalized. For each segment, the maximum size of its associated error rectangles has been listed in the third column. Notice that the error rectangle of the fourth segment is smaller than the previous one. This is due to orthogonalization. In fact, after orthogonalization, the third segment has changed its initial and final y-co-ordinates from 103 and 99 into 101, and its initial and final errors of 8.1 and 12.36 units into 10.23. This error in the position of the third segment also becomes the larger side of the error rectangle associated to the initial position of the fourth segment, which grows it until 11.01 local units at its final point.

By analysing the previous table, we can observe that these nine segments are just covering portions of 5 different walls. This is due to their overlapping. In fact, the eighth and ninth segments are respectively included in the sixth and seventh segments. Therefore, since they are just adding error, we can set the maximum allowed error to be equal to 30 units (that is, 60 cm.). The next Figure 7.13 presents the resulting global map of including the same four partial maps with a maximum allowed error of 30 local units.

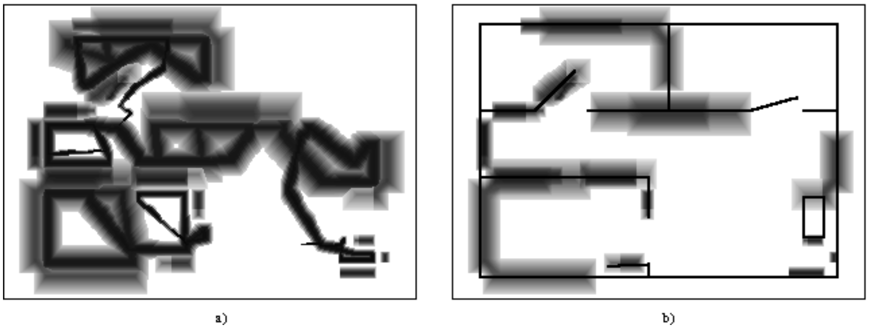


Figure 7.13: Global map considering information with a maximum error of 30 units. a) Including trajectory information. b) Real environment over the global map wall information.

In fact, the user of the application does not need to take into account very accurate information in order to decide what is the balance between error and coverage. Just by looking at the representation of the map in the screen, he or she can conclude that image b) in Figure 7.13 covers nearly the same walls than image f) in the previous Figure 7.12. And regarding the error associated to the information, it is equally simple to recognise the decrease in the error. Next Table 7.5 contains the details of the new global map.

Partial map information						**Global map**		
#	#t. seg.	total dist.	#w. seg.	#s.pt.	wall dist	wall dist	#s.pt.	coverage
1	29	505.7	7	3	267.2	237.3	2	16.2%
2	37	524.6	6	2	140.7	442.8	4	30.1%
3	33	543	7	2	262.1	670.8	6	45.7%
4	31	528.4	5	2	216.8	838.6	7	57.1%

Table 7.5: Global map analysis reducing the maximum allowed error to 30 units (60 cm.).

Finally, just briefly comment that we can observe form the previous images that the computed wall positions do not correspond exactly with the walls in the environment (despite the fact that segment orthogonalization corrects significantly their positions). This is mainly due to robot errors and that the host cannot know the exact distance between the robot and the followed wall. Although it is not realistic, we compute the wall segment at a constant distance to the trajectory segment. This distance is taken to be equal to the distance of detection of the infrared robot sensors. The other reason is the change in the robot orientation during wall following. This change cannot be reflected into the partial map, which only contains information about the positions where the robot started and ended the wall following task.

Chapter 8

Map Refinement

The previous chapter presents the way by which a host computer generates maps of unknown environments that have been previously explored by a troop of robots. The resulting map is a grid representation to which partial maps information has been added. Partial maps contain robots' trajectory segments that are used as evidences of the position of walls and free-space areas. These evidences are translated into possibility and necessity degrees. This assignment does not represent segments explicitly. Nevertheless, there is implicit information in the grid that allows us to treat adjacent cells as belonging to the same portion of a wall. This chapter describes how this implicit information can be used to refine the global map. On the one hand, wall information can be grouped in segments that can be extended under certain circumstances. And, on the other hand, wall information can be grouped into polygon-shaped obstacles that allow to plan paths towards less explored areas.

8.1 Map Treatment: Wall Extension

We have already described how robots move pseudo-randomly in unknown environments. If while exploring, a robot detects a wall —or obstacle—, it follows its contour along a random distance and marks in its partial map the corresponding trajectory segment as wall detection (so that the host can also compute the position of the corresponding wall segment). Following a wall along a random distance implies that the probability for a robot to leave a wall before reaching its end is proportional to this probability. As the fifth chapter describes, this probability corresponds to the turning probability, which has been defined by the user for each robot. Random distances have been defined to belong to a prefixed interval in order to guarantee upper and lower distance bounds. Therefore, unless the environment has extremely short walls, robots leave the walls before reaching their ends. (We have already commented that this is done in order to increase the number of discovered features, i.e., to look for other detections, and to avoid robots going around the same obstacles uninterruptedly). The host can distinguish that a robot has reached the end of a wall —or obstacle side— by means of the presence of a *singular point* label associated with the final point of the segment.

Both wall and trajectory segments are discretized into grid positions, and for each position, we have seen that necessity and possibility values are assigned to cells in the grid. When the position corresponds to a singular point, the cells are marked accordingly. In this manner, although the segments are not explicitly represented as such in the grid, the host can treat neighbour cells with positive necessity values and equal orientation labels as belonging to the same piece of wall or object. And obviously, if this piece of wall or object has cells containing singular point labels, it means that there is a discontinuity in its shape. (It could correspond to a corner, an open door, etc.). In the sequel, we will refer to these cell groups as wall segments because is for this kind of environmental features that the extension we propose makes more sense. Following this grouping cells idea, we can think of the global map as an implicit representation of wall segments and trajectories.

The obtained map is as reliable as the information error allows, but it is also relatively limited. The previous chapter has shown the results of map generation in terms of wall coverage, and although they cannot be considered as being poor, these results can be improved. Actually, we can ensure that wall segments without singular points correspond to longer walls, and therefore, we can extend them. However, there is not enough information in the map to know the magnitude of the extension. In the absence of the knowledge of real wall lengths, we use some criteria to limit the extension of wall segments. First, since trajectories represent free space, they are used as extension bounds (see Figure 8.1). And secondly, it seems reasonable to stop extending a wall segment when it meets either another detected wall segment or another segment extension. In this later case, if the meeting wall segments have the same direction, then they can be considered as being part of the same wall. On the contrary, if they have perpendicular directions, the direct interpretation is to consider that they form a corner, so that their junction can be labelled as *hypothetical singular point* (see Figure 8.2).

Extension is nothing but a conjecture, there are no evidences to support it. As a consequence, to be conservative, the host only extends orthogonal wall segments (i.e., vertical or horizontal). This is because, on the one hand, it is more likely that orthogonal wall segments actually correspond to real walls, and since walls are in average longer than the distance they have been followed, the extension is appropriate. On the other hand, oblique features usually correspond to doors or other less usual objects. Thus, considering that it is more difficult to predict the real shape of oblique features, it is safer not to extend them.

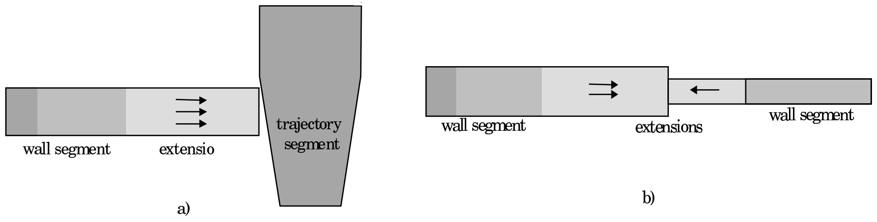


Figure 8.1: a) Wall segment extension stopped by a trajectory segment (the left side of the segment does not extend because of the presence of a singular point); b) Two wall extensions in the same direction meet to form a single wall.

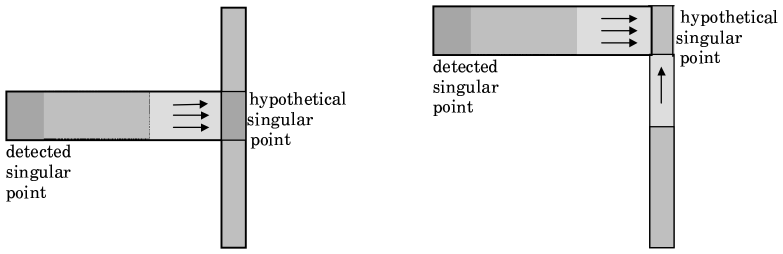


Figure 8.2: Segment meetings with different orientations that define hypothetical singular points.

8.1.1 Implementation

Extension is done locally by propagating low constant certainty values of occupation for those cells in the segment extremes. These low necessity values are set to 0.1 in order to reflect that they are just assumptions and do not correspond to actual robot detections.

As we have already said, segments are not explicitly represented in the grid. Therefore, the extension algorithm must search trough the grid those cells belonging to the extremes of a group of cells that can be considered as a wall. These cells are characterised by two conditions. On the one hand, they must correspond to a discontinuity in the necessity values. And, on the other hand, the discontinuity must be in the direction of the wall segment. In algorithmic terms, this means that this cell must have a positive necessity value, and one of its neighbours in the direction of its orientation must have a zero necessity value. In this manner, if for example there is one cell with positive necessity and horizontal orientation, then either the cell on its right or the one on its left must have zero necessity value (up and down respectively for vertical segments). Each time a cell with one neighbour with zero necessity value is localised, it is necessary to check if it can be extended (that is, if the stopping conditions do not apply). If it is the case, the neighbour is marked for extension (with an 'extendable' mark), otherwise, the cell is marked as 'non-extendable'. In the extension algorithm, once all cells have been checked, it assigns 0.1 necessity values to cells with 'extendable' marks (and the 'extendable' marks are then removed).

The user establishes the maximum length that segments can be extended. (Obviously, if the maximum is larger than the dimensions of the grid the system just extends as far as it is possible). Extension is done "one cell at a time" in order to extend by the same distance all segments in the grid. Therefore, a unitary extension is done over the grid a number of times equal to the maximum extension length (obviously, the process also ends if no cell is extended). As a consequence, extendable marks need to be updated for each unitary extension: a cell in the extreme does not longer belongs to the extreme once its neighbour changes its necessity value from zero to a positive value. On the contrary, 'non-extendable' marks last for the whole process.

The next pseudo-code corresponds to the overall extension process:

```
Grid_Extension( extension_length )
{  k=0, end=0
   Repeat for (k<extension_length  &  end=0)
   {  Repeat For Each cell(i, j) in the grid
      {   if( cell(i, j).N>0 & cell(i, j).extension_mark=0 & cell(i, j).sing_pt='n')
               determine_and_assign_marks(i,j)
      }
      Repeat For Each cell (i,j) in the grid
      {  if(cell(i, j).extension_mark='y')
         {  cell(i, j).N=0.1
            cell(i, j).extension_mark=0
            end=0
         }
      }
      k=k+1
   }
   remove_extension_marks_in_the_grid()
}
```

The determine_and_assign_marks() function is in charge of determining the neighbours of a cell, and for each cell, it assigns an 'extendable' mark to a neighbour cell or a 'non-extendable' mark to the cell. This assignment follows the idea of limiting wall extension by robot trajectories. A neighbour is considered to be extendable only if its necessity is zero and its possibility value is bigger than the possibility value of the cell being considered. Or, in other words, if the evidence of free-space in the neighbour is smaller than the evidence of wall in the cell. Otherwise, the cell and all cells in this segment extreme are marked as 'non-extendable'.

Additionally, if a neighbour corresponds to an extension that is perpendicular to the current cell orientation, then a hypothetical singular point is labelled for all the cells that belong to the intersection of the two wall segment extensions.

8.1.2 Results

Figure 7.12 in Section 7.5.2 shows the exploration of four robots and the incremental generation of the corresponding global map. Here, we will use the same maps to illustrate the results of the extension process.

The image a) in the next Figure 8.3 shows the current global map after the addition of the information of the first robot's partial map. As the first row at Table 8.1 indicates, a total length of 237.3 units corresponds to walls that have been actually followed by the first robot (and this implies a wall coverage of 16.2%). If then, after the addition of the first partial map, the user chooses to extent the walls of the global map, then the system will generate the map shown at image b) in Figure 8.3. As we can see in this image, extension is only applied for segment ends without singular points. In fact, two hypothetical singular points are defined when trying to extend the segments that appear on the left. As we can observe in the upper-right wall segment, when trying to extend a necessity value to a cell that does not have the default possibility value 1, possibility value comparisons are used to decide whether the extension comes from a stronger evidence than the free-space trajectory or not (that is, if the extension must be done). In the case of this wall segment, only its right extreme is extended, and since its extension does not encounter any other information, it propagates until the boundaries of the environment map. This extension covers the door gap that the environment has, and therefore, we consider that 20 units of the extension are incorrect. The second part of the first row in the same Table 8.1 contains the detailed values, which add a total of 112 extended units to the length of detected walls.

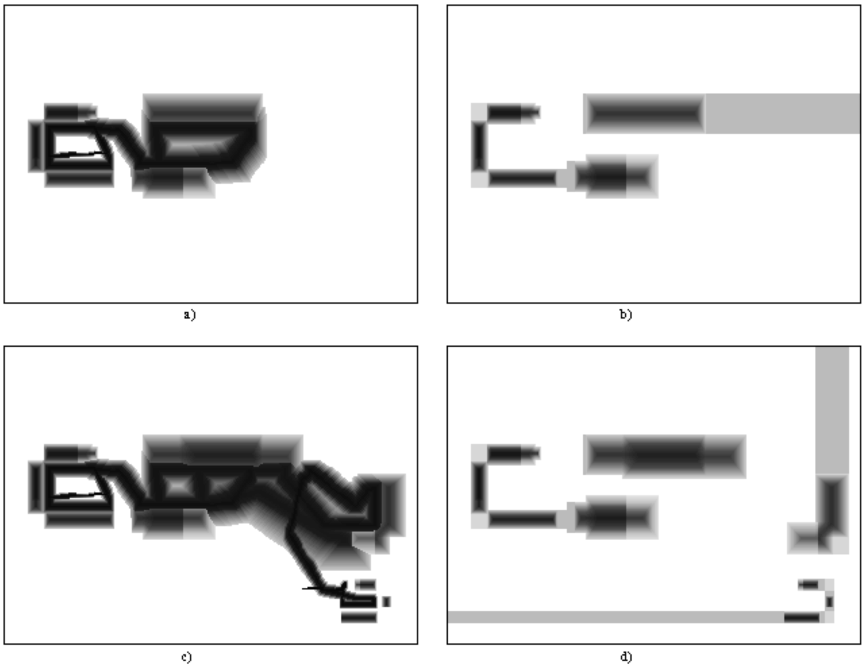


Figure 8.3: Extension of the maps from Figure 7.12.

The addition of the two consecutive partial maps (from two robots) yields to the situation shown in Figure 8.3 c). If subsequently, the user chooses the extension option in the host application, then the map in d) will appear. In this case, since there is more trajectory information, the extension hardly propagates necessity values in the inner area of the environment. As we can observe at the bottom of the image, a short wall following can be extended to delimit a complete wall in the limit of the environment. Obviously, as the second row in the table details, this kind of extension increases significantly the wall coverage without including, in that case, erroneous wall representations.

Global Map				**Extended Global Map**					
#	wall	s.pt	coverage	tot. ext.	cor. ext	n-c ext	s.pt	cor. cov	n-c. cov
1	237.3	2	16.2%	349.3	329.3	20	4	22.4%	1.4%
2	442.8	4	30.1%	808.8	808.8	0	9	55.1%	0%
3	670.8	6	45.7%	900.8	850.8	50	12	57.9%	3.4%
4	838.6	7	57.1%	1098.6	1048.6	50	15	71.4%	3.4%

Table 8.1: Extension analysis over the sequence of stages of the incremental generation of the global map. The abbreviated column names mean: '#' = number of

partial map; ‘wall’ = length of followed walls; ‘s.pt.’ = number of detected singular points; ‘coverage’ = wall coverage with respect to the ‘wall’ column; ‘tot. ext.’ = total length of the wall segments in the extended map; ‘cor. ext.’ = total length of correct wall segments; ‘n-c ext.’ = non-correct wall extension; ‘s.pt.’ = number of singular points; ‘cor. cov.’ = correct coverage; ‘n-c cov’ = non-correct coverage.

Image e) in the next figure represents the global map after including the partial maps of three robots. In this case, some of the walls followed by the third robot were already considered by the previous extension, and therefore, the wall coverage increase is not as large as the previous one (see third row in the table).

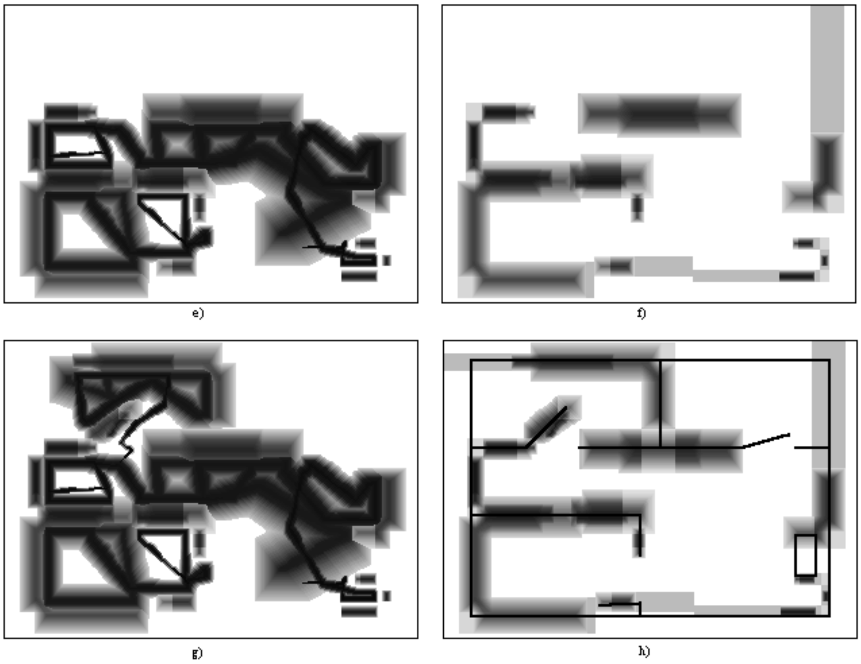


Figure 8.4: Continuation of the previous Figure 8.3: e) global map of three partial maps; f) its extension; g) final global map; h) extended final global map (the real environment has been included in order to evaluate the results).

And finally, the last images g) and h) in Figure 8.4 show the complete global map. In this case, the extension process increments the wall coverage by 14.3% and the length of the wrong assumptions represents a 3.4% of the total length of real walls. Obviously, the effect of the extension process decreases with the arrival of more information because the propagation of necessity values encounters more limits, however, this also implies that the risk of making wrong assumptions about wall positions also decreases.

The previous example has been used to illustrate how extension increases the map coverage of the walls in the environment. However, extension is based on environment assumptions, and therefore, its results not only depend on the robot exploration but also in the distribution of the features in the environment. Basically, the assumptions about the environment are:

- First, since extension is not applied to oblique features, the environment is assumed to be mostly orthogonal, and
- Second, walls must be significantly longer than the average followed length. That is, the longer the walls, the more sense extension has.

In this manner, the extension results will strongly depend on how the environment fulfils these two assumptions. We present now an additional example of environment (see Figure 8.5) whose distribution of features is less predictable than the one in the previous example. As we can observe, none of the two assumptions are properly fulfilled.

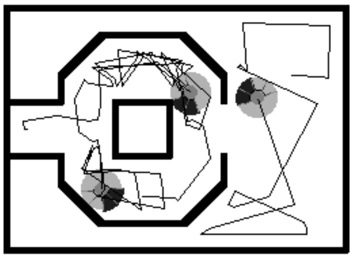


Figure 8.5: Exploration of an unpredictable environment by three robots.

Figure 8.5 above, shows the exploration trajectories of three robots. Their initial positions are (35, 110), (120, 60), and (230, 150) respectively, and their turning and left/right turning probabilities have been respectively assigned the following values: 0.25 and 0.55 for the first robot, 0.2 and 0.45 for the second robot, and 0.15 and 0.3 for the last robot (notice that, since its turning probability is smaller than the probabilities of the other robots, it performs longer displacements without changing its direction).

The resulting maps and their extensions are shown in Figure 8.6 below, and the coverage percentage is listed in the subsequent Table 8.2. As we can observe, the extension in this unpredictable environment topology gives a significant amount of erroneous assumptions. Once again, this errors are dependent of the amount of free-space trajectory information that limits the extension.

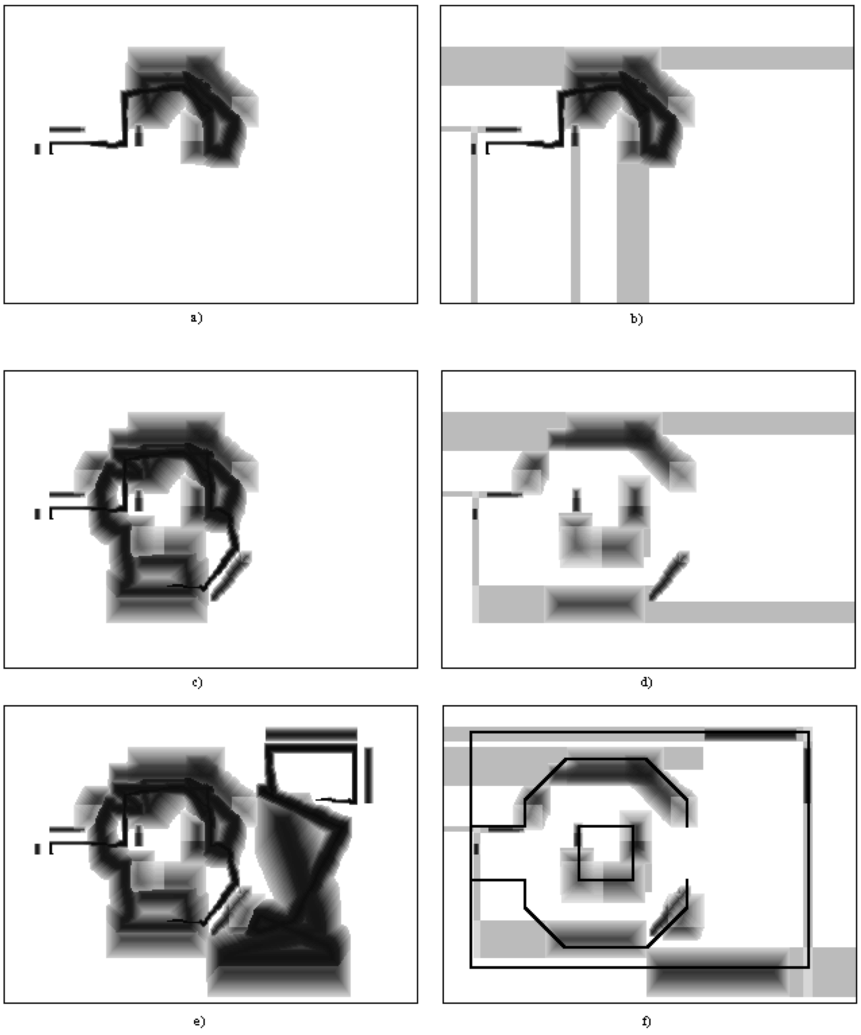


Figure 8.6: a) c) and e) Incremental generation of the global map of the environment in the previous Figure 8.5. b), d) and f) correspond to their extensions. In f) the real environment has been included in order to evaluate the results.

Global Map				**Extended Global Map**					
#	wall	s.pt	coverage	tot. ext.	cor. ext	n-c ext	s.pt	cor. cov	n-c. cov
1	161.9	4	12.1%	729.9	299.9	430	5	22.5%	32.2%
2	376.6	6	28.2%	806.6	476.6	330	8	35.7%	24.7%
3	591.6	6	44.3%	1204.6	974.6	230	10	73.1%	17.2%

Table 8.2: Extension analysis over the sequence of stages of the incremental generation of the global map. The abbreviated column names are equal to the ones in the previous Table 8.2.

Wall extension has two main advantages: first, it increases the coverage of the real environment and second, the planning over the resulting maps give more conservative and safer paths than the ones obtained considering just detected features. As we will see in the following section, these paths are computed from a visibility graph of the free space and guide robots towards less explored areas. In Section 8.3 we will see that, when robots follow paths computed from expanded maps, the use of reactivity to avoid unknown obstacles is reduced.

8.2 Path Planning

After the robot exploration and the subsequent global map generation it is possible that certain areas of the environment have been poorly covered. In the literature, the borders that separate already explored areas from non visited areas are known as "frontiers". The work done by Yamauchi (Yamauchi 98) is more focused on localising them in the map than on the approach to plan the path towards them. Yamauchi uses evidential grids and define a frontier as the border between a known open space that yields to a unknown space. In our approach, is the user of the host application who chooses an initial and a final position for the robot to approach less explored areas. These positions are defined by clicking on the map screen, and afterwards, the host generates a path between these two positions.

In the present section we will see how the map representation on the grid is transformed into a graph representing free space. And once the graph is build, a path search algorithm can be applied to obtain the shortest path in the graph.

8.2.1 Graph Extraction

The map generation process results in a representation of the environment global map that consists in a Possibility/Necessity grid. This representation is based on a discretization of the environment and is very useful to combine evidences. However, it is less suitable for planning paths in free space. As a consequence, the planing process in the host extracts, from the information in the grid, a graph representation of the environment that simplifies the search of the required path.

In the literature (Russell 95) there are different methods for obtaining a graph representation of the free space in an environment. We briefly describe here three of them:

- Cell decomposition. This method decompose free-space in convex regions (of a simple shape) called cells. From these free-space cells, an adjacency graph is built by considering cells as nodes and establishing arcs —or edges— between adjacent cells. In order to plan the path, the initial and final nodes are the ones that contain the initial and final positions respectively. The resulting path is a sequence of cells and is translated into a linear path by drawing lines between the central position of each cell and its boundary with the next cell.
- Voronoi Diagram. Trajectories in free-space are found so that they are optimal in terms of distance to obstacles (they maximise this distance). The graph is computed by identifying the points that are equidistant to the obstacles in their surroundings. In this manner, consecutive equidistant points define the arcs in the graph, and their bifurcations constitute the nodes.
- Visibility graph. This method assumes that the obstacles in the environment are polygons. The free-space graph is build by taking as nodes all the vertices of the polygons. Arcs join 'visible' nodes, meaning that it is possible to move from one node to the other following a straight line (without encountering obstacles). Such a graph contains the shorter path between two points whenever the obstacles in the environment are polygonal. If the points do not belong to the graph, they are included as additional nodes and their corresponding arcs are also established. The resulting graph guarantees the completeness (i.e., to find a path whenever it exists) of the algorithm that searches the shortest path between the initial and final points. Next Figure 8.7 shows an example of Visibility Graph.

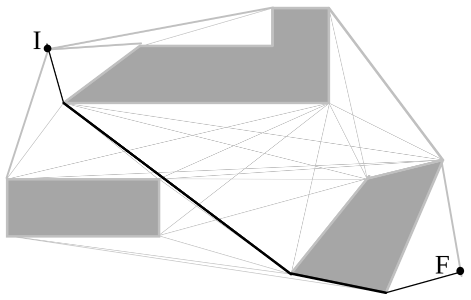


Figure 8.7: Example of Visibility graph of an environment of three polygonal obstacles. In black, the shortest path between the initial I and final F points.

Cell decomposition seems to be closer to our grid representation: we could easily consider as nodes our cells with zero necessity values and build the adjacency graph. However, this would yield to a very large number of nodes. Moreover, this method is sound but not complete (i.e., it produces a safe path but may not always find one if it exists). On the contrary, Voronoi diagrams and Visibility Graphs provide a reduced graph description of free space —named skeleton— that avoids an explicit description of the free-space boundaries. They are known as skeletonization methods and guarantee completeness under certain circumstances (the skeleton must have a single connected region and it must be connectable to the initial and final points in free-space).

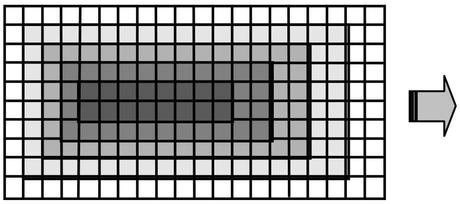
Voronoi diagrams have been widely used in path planning (Ilari 90) when the map contains a complete representation of the environment. In this manner, since it maximises the distance to the obstacles, it yields to the safest paths. In our case, the map information is not complete and we want the host to compute paths towards specific unexplored areas through known and secure areas. However, this method would guide our robots through less explored areas (because the lack of wall information would be interpreted as larger distances to the detected objects) and therefore, we decided to use the visibility graph method. Another reason for choosing it is its optimality in terms of length. Our robots have very limited energy resources and the localisation error increases with the covered distance, therefore, to give the shortest path is a key aspect to reduce as much as possible the waste of batteries and the odometry error. Furthermore it presents two additional advantages. First, the resulting paths are sequences of adjacent rectilinear displacements, and therefore, it accommodates perfectly the movements of our robots. And second, since the resulting paths force robots to reach previously detected obstacle vertices, they could be used as landmarks in the path following process.

As we have already said, the Visibility graph method can guarantee the generation of the shorter path only if the considered environment information is complete. Although this does not apply to our case, it is still useful for our robots when navigating inside the environment: they can use this path as a guide that avoids the previously detected obstacles and they can apply reactivity when encountering undetected obstacles. Obviously, this means a loss of optimality when navigating in the environment. Nevertheless, an optimal path that considers non-complete information may probably be the best available heuristic that a robot can use to guide its navigation.

Graph Building by means of Image Analysis Techniques

Before applying the Visibility Graph method, it is necessary to identify the vertices of the polygonal obstacles in the global map grid. In order to find them, the application in the host computer adapts two widely used image processing techniques: thresholding (Russ 95) and template matching (Parker 94).

Thresholding is a method that filters grey values in a B/W image and obtains binary images (whose treatment is simpler). The filtering consists in using a grey value that acts as a threshold assigning a 0 or a 1 value to each pixel. In our case, we assign 1's to cells having positive necessity values and 0 to the rest of cells. (see Figure 8.8).



```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 8.8: Wall segment thresholding to obtain a binary image.

On the resulting binary map, we use template matching to recognise polygon vertices. This method consists in representing a specific feature in a template so that when comparing pixels in the image with this template, the feature will be found whenever they coincide. Figure 8.9 shows eight templates that identify the vertices in the given polygon.

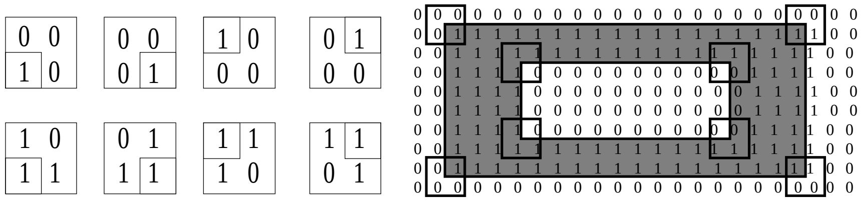


Figure 8.9: Binary templates that determine 8 different kinds of vertices in the polygon shown next to them.

Once we have identified a vertex position in the binary grid, the graph building method checks if it is possible for a robot to be positioned in front of the vertex position (that is, if there is a free-space square area adjacent to that position large enough to include robot). If it is the case, then the central position of the square is taken as a graph node. Graph arcs are afterwards computed to join visible nodes. The number of visibility comparisons between polygon vertices can be reduced by considering their characteristics. For instance, we can ensure that the concave corner in obstacle A at Figure 8.10 can only possibly be visible for those other vertices having lower or equal 'x' and 'y' co-ordinates (so that the number of comparisons is reduced from 13 to 6). In our grid, visibility is easily checked by drawing a straight line on the grid without encountering any occupied cell. The distances between vertices are included in the visibility graph as labels associated to the arcs that represent traversability costs.

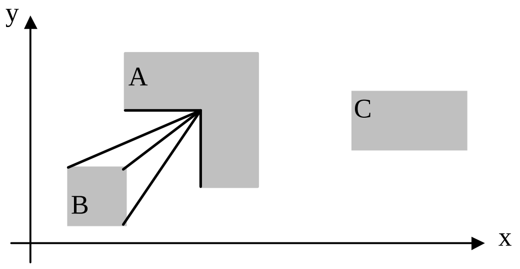


Figure 8.10: Visibility arcs —or edges— for a concave vertex in A.

We end the graph building process by including two additional nodes to the graph: the initial and final positions of the path (they are added as single 1's in the binary grid).

8.2.2 Computing the Shortest Path in the Visibility Graph

Once the visibility graph has been generated (including the initial and final points that the user has chosen), the path planning method in the host application searches the shortest path in the graph that joins these two points. Since the costs of the arcs in our graph represent the distance between nodes, our shortest path search is equivalent to searching the path with minimum cost.

The cost of a path is defined as the sum of the costs of all the arcs along the path. In this manner, we search the path going from the initial node n_i to the final node n_f, such that its cost c is minimum. The search can be done based on different strategies (Pearl 84): irrevocable control (as Hill-Climbing), systematic search (Depth-First Search or Breadth-First Search), and informed control, which uses partial information about the problem domain to guide the search to increase the search efficiency. This information is called heuristic knowledge (heuristics) and is used to evaluate which node should the search consider next. We have applied the widely known A* algorithm that belongs to the family of informed strategies. Although we are not going to detail it here (a complete description of the algorithm and its properties can be found in (Pearl 84)), we introduce some notation and definitions in order to prove that the A* properties are fulfilled when the algorithm is applied to our specific problem domain.

Notation:

$k(n_i, n_j)$: cost of the optimal path between two nodes n_i and n_j,

$h^*(n)$: cost of the optimal path from a node n to the final node n_f,

$g(n)$: cost of the current path from the initial node n_i to n,

$h(n)$: heuristic function that estimates $h^*(n)$,

$f(n) = g(n) + h(n)$: evaluation function that is used to select the next node to consider in the search algorithm.

Definitions:

Completeness: an algorithm is complete if it ends with a solution whenever there is one.

If, in addition, the solution is optimal, then the algorithm is considered to be *admissible*.

An algorithm is *optimal* when it has the maximum efficiency for finding the solution.

An heuristic function h is *admissible* if:

$$h(n) \le h^*(n) \quad \forall n,$$

and h is *consistent* if and only if it fulfils the triangular inequality:

$$h(n) \le k(n, n') + \mathrm{h}(n') \quad \forall n, n'$$

Properties:

In finite graphs (graphs were each node has a finite branching degree or number of successors), the A* algorithm always ends and it is complete.

If the heuristic function h used in A* is admissible, then, A* is admissible as well.

A consistent h guarantees that when a node n is treated (it is expanded, so that its successors will be considered), the search has already found the optimal path from the initial node to this node n, and therefore, this portion of the optimal path will not be changed.

In our case, the heuristic function has been taken to be the Euclidean distance between a given node n with co-ordinates (x, y) and the final node $n_f = (x_f, y_f)$:

$$h(n) = \sqrt{(x_f - x)^2 + (y_f - y)^2}$$

And therefore, we can prove that:

- Our heuristic function is admissible because we can ensure that the real cost $c(n, n_f)$ will always be greater (or equal, in case there are no obstacles in their way) than the Euclidean distance.

- Since h is admissible, the search is also admissible so that we can guarantee that it will end with the optimal solution: the shortest path.
- The search is optimal (in terms of efficiency of obtaining the solution) due to that the Euclidean distance heuristic also fulfils the triangular inequality. (This optimality proof simplifies the implementation).

8.2.3 Results

The following Figure 8.11 shows a global map that contains the information gathered by four robots after exploring an environment with granularity 1 unit (which equals to 1cm). Over this global map, the user has chosen an initial and a final position located at (476, 24) and (316, 384) respectively. These positions appear in the map labelled by their corresponding I and F initial letters. Afterwards, when the user chooses the Path Planning option, the Visibility graph is generated and the shortest path is searched. The process ends with by displaying the resulting path that appears in blue.

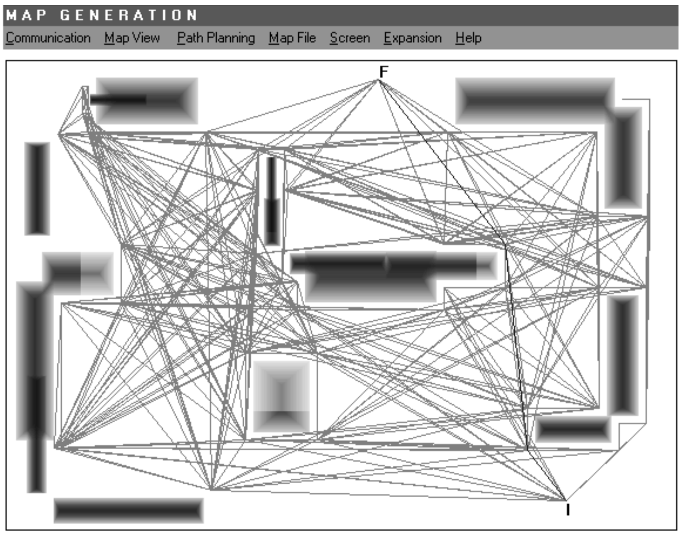


Figure 8.11: Visibility graph generated for a global map. Nodes are free-space positions that are safely away from corners. Arcs appear in grey, and the obtained path between I and F in blue.

We have already explained that only those vertices having enough free-space in front of them are included as nodes in the graph. This is computed by checking if a 25×25 square of cells has zero necessity values, and if it happens to be the case, then the (12, 12) position inside this square is included into the graph as a new node. In this manner, the node represents a position that is almost 17 units distant to the corner in an oblique direction. (notice also that most of the vertices close to the borders do not have nodes because their distance to the environment borders is smaller than 25).

The obtained path is displayed on the host screen but and saved in a file so that a robot can use it as a guide to go from the initial position to the final position in the environment. In this example, the resulting path has four points: (476, 24), (443, 67), (424, 243), and (316, 384) and a total length of 409 units (that this case correspond to 409 cm.).

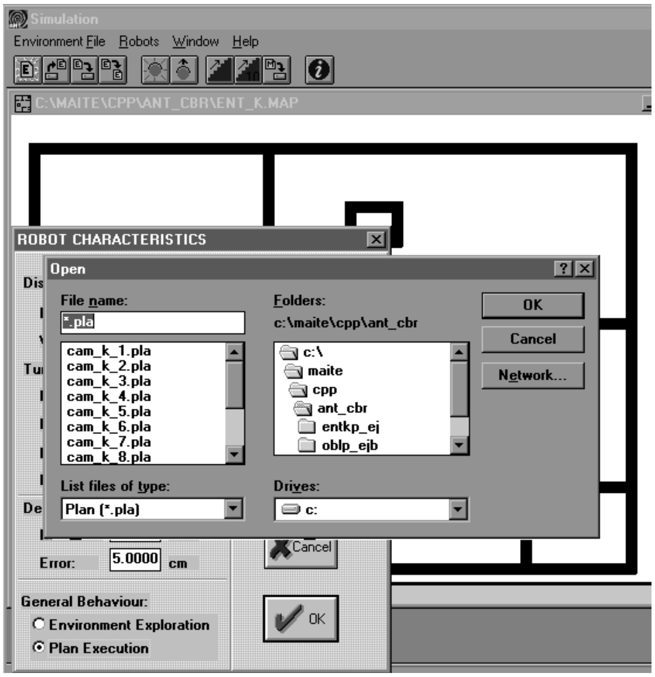
In the previous subsection we have seen that the A* algorithm ensures that the obtained path is the shortest in the visibility graph that joins the initial and final points. In addition, visibility graph also ensures that its representation contains the shortest path between any two points in the free-space. Nevertheless, we cannot conclude that our approach actually gives the shortest path but a very close one. This is so because we have slightly modified the visibility graph representation in order to obtain safer paths than the ones that join the obstacle vertices. This node modification may also imply the inability of computing paths that should go trough narrow walls or passageways. However, if this is the case, the robot could neither pass through, and therefore, not giving the path just models a conservative approach.

8.3 Path Following

Robots use the paths that have been planned at the computer host to reach less explored areas. In fact, the simulator system that has been used for robot exploration, simulates also robots following given paths.

Most of the simulator implementation issues explained in Chapter 5 also apply when the robots perform a path following strategy. Obviously, a robot that follows a path needs its specification. The path is represented by intermediate points that the robot must reach consecutively when going from its initial position to a final one.

Figure 8.12 shows the dialog boxes that appear in the simulator when the user defines a new robot. They are equivalent to the dialogs that appeared in Figure 5.2 (page 106) when adding an exploratory robot. The error values are exactly the same, and the general behaviour option is now selected to be “Path Following”. Over the dialog of robot characteristics, there is a dialog that asks the file containing the planned path that the robot will follow. These dialogs will locate a new robot in the initial position of the path, and if afterwards, the user chooses the start robot option, the robot will follow the path until it reaches the last position in the path (that is, the final desired position).



C:\MAITE\CPP\ANT_CBR\ENT_K.MAP

File name: *.pla

cam_k_1.pla
cam_k_2.pla
cam_k_3.pla
cam_k_4.pla
cam_k_5.pla
cam_k_6.pla
cam_k_7.pla
cam_k_8.pla

Folders: c:\maite\cpp\ant_cbr

c:\
maite
cpp
ant_cbr
entkp_ej
oblp_ejb

List files of type: Plan (*.pla)

Drives: c:

Error: 5.0000 cm

General Behaviour:
Environment Exploration
[✓ Plan Execution]

Figure 8.12: Adding a new robot to the environment that will follow a planned path.

8.3.1 Behaviour-based Architecture for Path Following

Chapter 6 describes a Behaviour-based architecture for environment exploration. Here, although we need to specify a new strategy for a new task, we can adapt most of the ideas applied to the random exploration because we still need to follow walls in order to go around non previously detected obstacles. In this manner, our aim is to illustrate the idea that the proposed basic behaviours are definitely specialised for a task, but they can also be easily adapted to other tasks.

Just by looking at the Figure 8.13 we can easily notice the similarities between this automaton defined for the path following strategy and that automaton defined in chapter 6 (see Figure 6.1) for the exploration task.

Their main difference is that random probability decisions are substituted for goal oriented decisions. As a consequence, the random walk behaviour has been replaced by a *directed walk behaviour*, which is in charge of directing the robot towards the next intermediate position in the path. Moreover, these goal oriented decisions also imply some other changes in the remaining behaviours.

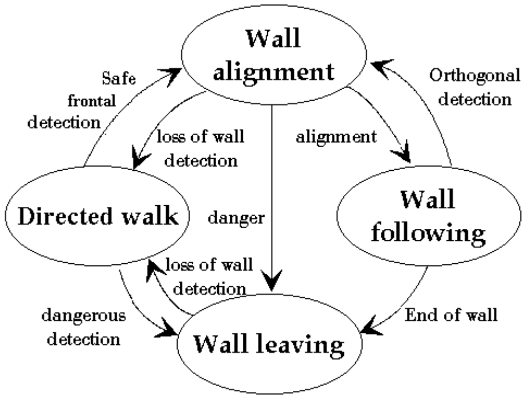


Figure 8.13: Path following automaton.

Since these basic behaviours have the same structure as those of exploration, we point the reader to Section 6.1 to know how behaviours contain sets of If-Then rules that generate and supervise actions. We neither describe here these rule sets but the intuition of their performance :

- Under normal circumstances, the *directed walk* behaviour computes the distance and orientation that the robot must follow in order to reach the next intermediate point.
- Since the given path only considers previously detected obstacles, it can be the case that the robot encounters an unexpected obstacle obstructing its way. In such a case, the remaining behaviours co-ordinate themselves to avoid the obstacle. This co-ordination performs a reactive strategy that allows the robot to deal with partially known environments.

We briefly describe now how the elementary behaviours are adapted in order to fulfil the path following task and include reactivity (notice that a full description of the behaviours appears in Section 1.1).

As we have already said, the *directed walk* behaviour directs the robot towards consecutive subgoal positions while no obstacle obstructs its way, that is, there is not frontal detection that may prevent the robot to maintain its orientation and displacement. In terms of sensor readings, this is equivalent to say that:

"the robot keeps following the path trajectory while there is not danger of collision, the frontal sensor does not detect anything, and there are not 'near' detections on the two oblique sensors"

In this manner, this behaviour ignores any other sensor reading, focusing only in those readings that can cause problems. Some of these problems are caused by obstacles that can be properly faced and surrounded, whilst others will require to escape from dangerous readings.

For the former case, a safe detection will cause the activation of the wall alignment behaviour. As we know, this behaviour forces the robot to face the wall (or obstacle side), turn ±90° in order to become parallel to the wall, and activate the wall following behaviour. Since there is no information about the shape of the obstacle, the next subgoal position constitutes the only thing that can be considered when determining the sign of the turning angle. In this manner, the robot will turn 90 degrees on its right or on its left depending on the direction that may give the shortest path towards the subgoal.

When the wall following behaviour is active, it controls the robot until the end of the wall is reached. Initially, it computes an initial distance to cover in the direction that the wall alignment behaviour decided. But if after having followed this distance, the wall end has not been reached yet, then the robot will try to follow the wall in the opposite direction for a distance that is twice the previous one.

Under normal circumstances, wall following will end and switch the robot control to the wall leaving behaviour. The natural sequence of behaviour activations will consists on: first, the wall leaving behaviour avoids the corner by moving forward a fixed distance d (see Sect. 6.2.1 for details), and second, the directed walk behaviour makes the robot to turn around the corner before resuming the robot movement towards the subgoal. In order to turn around the corner, the directed walk behaviour needs to use the information of action history to on know on which side the wall following was done. Once it finds the last wall detection in the history of actions, it forces the robot to turn 90 degrees towards the same side. The robot will finish the turning by covering the same fixed distance d in this direction.

From these comments, we can observe that this directed walk behaviour makes the difference between the exploration strategy and the path following. And this is not only because subgoal positions replace random computations, but also because the directed walk behaviour increases its priority of activation. This behaviour switches the activation of another behaviour only when it is absolutely necessary (when a frontal obstacle blocks its path). In this manner, even a near lateral detection does not activate the wall leaving behaviour if the sensor at the front does not have a reading. In such circumstances, the directed walk behaviour slightly modifies its orientation trying to avoid the lateral detection without deviating too much form the path it follows.

As in exploration, the wall leaving behaviour faces any problematic situation and causes the robot to move backwards along a fixed distance following a random orientation. This is done with the aim of reaching a new position where the directed walk behaviour can be active again. Obviously, if it is not the case, it will try to escape again or to surround the obstacle.

Finally, the ending condition is satisfied when the last point in the path has been reached. Nevertheless, it can be the case that the robot cannot reach this position. Hopefully, the robot will be moving while trying to reach it. This will imply an increase in the error rectangle associated to the robot position. The ending condition that is used under these circumstances is that the robot stops when its error rectangle includes the goal position. (The same condition applies for any intermediate subgoal).

8.3.2 Results

The next Figure 8.14 shows an example of path following with reactivity. In this example, the planned path did not take into account the presence of the rectangular obstacle, and therefore, it is specified as a rectilinear trajectory between the initial and final points. In this manner, the path that is given to the robot consists of the two positions that correspond to the small circles drawn in the environment. Their co-ordinates are respectively: (55, 20) and (60, 80) in local units.

Initially, as we can see in the first image of the figure, the robot is added into the environment at the initial position (55, 20). The first active behaviour is directed walk, which computes the distance and orientation of the action that the robot must execute in order to reach the goal position (see Sect. 5.3.2 to see the structure and execution of actions). During the execution of this action, a frontal detection occurs (see image 2 in the figure), and the action supervision rules of the behaviour activate the wall alignment behaviour. Once the robot is facing the wall (or side of the obstacle) properly, this behaviour decides to turn 90 degrees right because the goal position is on the right, and therefore, it is assumed that the right direction will likely yield to a shorter trajectory than the left direction. From this situation, which is depicted at the third image, the robot follows the wall on its left. From image 4 to 7, the wall following behaviour controls the robot and detects the end of the wall (see sequence of movements in images 5, 6, and 7 and further details in Sect. 6.2.3). Afterwards, images 8 and 9 show how the wall leaving behaviour corrects the robot deviation and moves it forward along 25 units to pass the corner safely. From this situation, the directed walk behaviour is again in control of the robot until the end of the path following task. Its first action is to turn left 90 degrees (see image 10) and cover 25 more units (image 11) in order to turn the corner. The next action is to move towards the goal position (image 12), and the robot executes it until a oblique 'near' detection forces the supervision rules to decide to deviate the current orientation by –7.5 degrees (see images 14 and 15). Finally, the robot does not find any other obstruction and can reach the goal (although, in fact, it ends at the position (65,77) due to the errors).

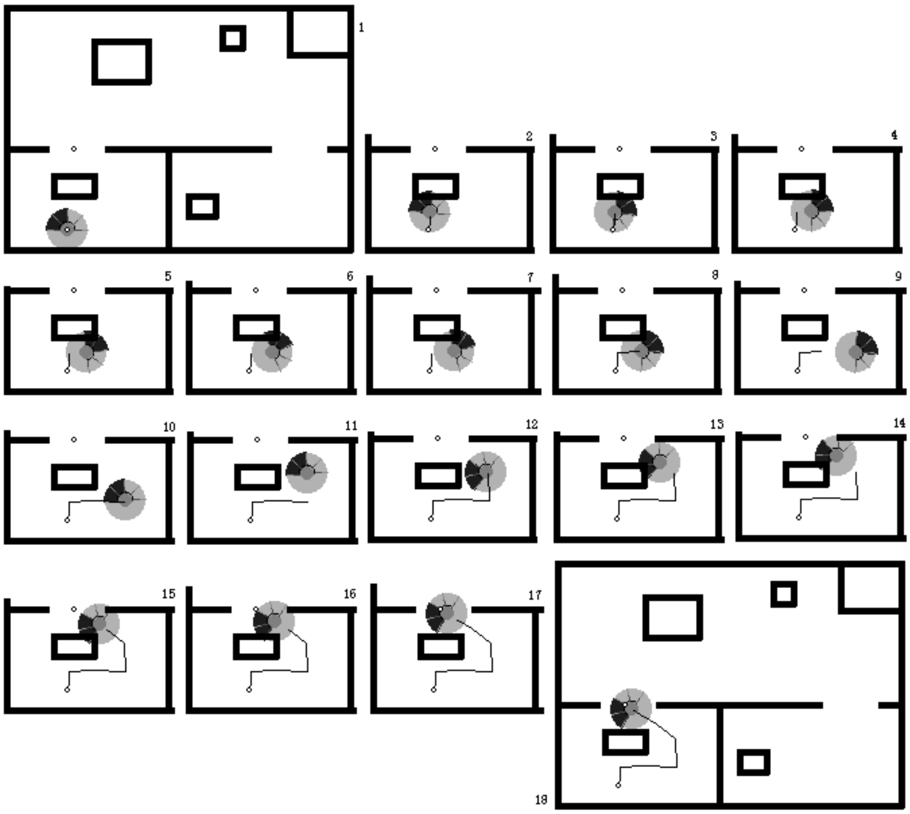


Figure 8.14: Example of path following with reactivity. 1) initial position (directed walk behaviour). 2) activation of wall alignment behaviour due to an obstacle obstructing the path. From 3) to 7): wall following and detection of the wall end. 8) and 9): wall leaving avoids the corner. From 10) to 18): directed walk drives robot until the goal position. In 10) and 11) it turns around the corner, in 12) and 13) goes towards the goal; 14) and 15) changes slightly the direction to avoid the obstacle; 16), 17) and 18) reaches the goal.

8.4 Conclusions

Up to this point, we have seen how the host generates a global map from an unknown environment and computes paths that can be used by robots to reach less explored areas. Obviously, we want these paths to be as accurate as possible. Otherwise, the robot would need to react against unexpected obstacles without knowing its shape. Reactivity is necessary for the robot to handle problematic situations. Nevertheless, since it is non-informed, it can yield to wrong decisions resulting in long trajectories that accumulate additional error. We conclude this second part by showing that the map extension process helps to generate more accurate paths that reduce the use of reactivity.

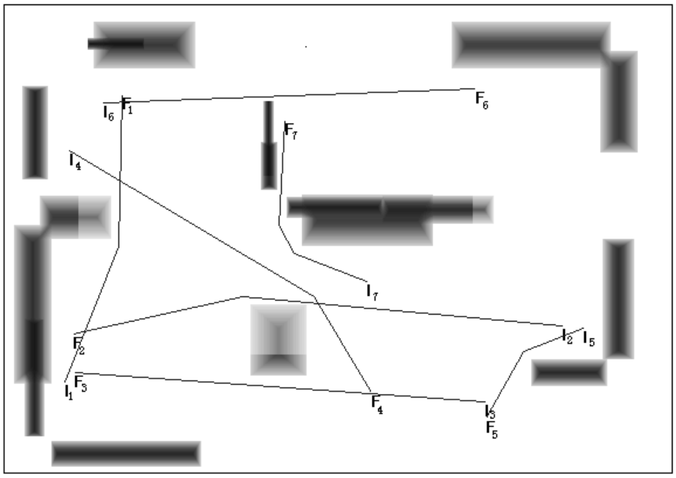


Figure 8.15: Seven different planned paths over the graph in Figure 8.11.

In Section 1.1, we have already commented that the extension process is purely based on the application of heuristics and, therefore, it is hard to evaluate how it helps in the coverage of the environment. Our aim here is to illustrate how extension helps in the generation of paths, and since its evaluation has the same difficulty, we do it by means of representative examples. We extend the example Figure 8.15, which gives a 45.3% of wall coverage. The extended map appears in Figure 8.16 and increases the wall coverage up to a 70.1% (this percentage only corresponds to correct coverage, an additional 6.4% has been incorrectly extended).

In order to compare the path planning in extended and non-extended maps, we have computed seven different paths for each map. In both maps, these paths have the same initial and goal position and are labelled with the same numbers. The specification of each path appears in the following Table 8.3. The first column of this table lists the path number, and the second and third columns contain the initial and final positions of each path. Comparing the paths in Figure 8.15 and Figure 8.16 we can observe that most of them describe different trajectories because they consider different maps. The fourth column shows the intermediate points between the initial and final position that result from the path planning over the non-extended map. The corresponding intermediate points for the extended map appear in three columns (7th, 8th, and 9th). As we can observe, extension results in longer paths that must avoid more features in the environment. The length of each path appears respectively in columns 5 and 9 for each map (non-extended and extended).

#	initial **I**i	final **F**i	non-ext.	dist	extended intermediate points			dist
1	(51,77)	(100,323)	(106,184)	255	(106,184)	–	–	255
2	(476,125)	(58,118)	(273,159)	422	(106,184)	–	–	422
3	(410,60)	(59,85)	–	352	(275,58)	(275,112)	(238,178)	430
4	(55,275)	(313,68)	(238,178)	340	(238,178)	(275,112)	(275,58)	359
5	(494,123)	(411,46)	(434,112)	120	(275,112)	(275,58)	–	358
6	(84,316)	(402,328)	–	318	(238,178)	(433,197)	–	512
7	(309,163)	(239,301)	(238,202)	183	(433,197)	(433,252)	–	353

Table 8.3: Specification of the seven paths in the non-extended map (see Figure 8.15) and in the extended map (Figure 8.16).

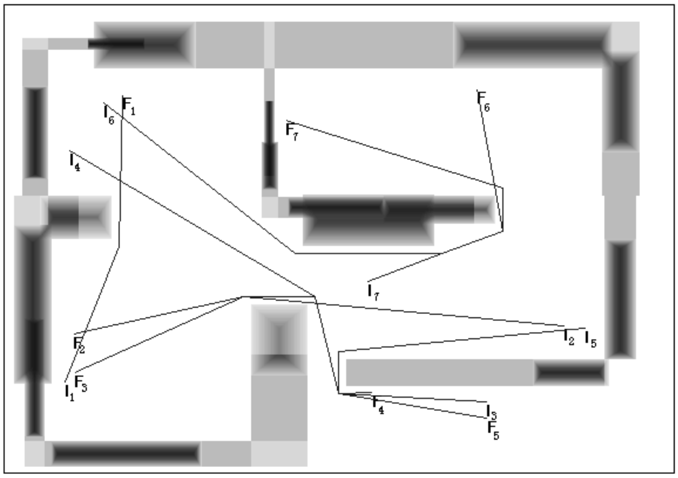


Figure 8.16: The seven paths connecting the same initial and goal positions of Figure 8.15 over the extended map.

The following figures (from Figure 8.17 to Figure 8.22) depict the trajectories actually followed by the robots while trying to follow the planned paths to reach the goal positions. Small circles along these trajectories denote the positions where robots reach each intermediate point (including the initial and final points).

We distinguish three different kinds of paths depending on the map area on which they are planned:

- If both areas, the one at the non-extended map and the other at the extended map, contain the same information, then the paths are obviously equal and the robots perform equally when following them. (The first and second paths correspond to this situation).
- If the area in the extended map has more correct wall information than the area at the non-extended map, then, the extended map yields to longer paths that are more realistic than the paths based on non-extended maps. Therefore, correct expansion means less use of reactivity.
- If the area in the extended map has incorrect extensions, then, the resulting path from the extended map is longer than the path from the non-extended. Nevertheless, paths in the extended maps are more conservative, because they tend to coincide with previous robot trajectories.

However, under certain circumstances, an incorrect extension can also be useful if it closes narrow gaps. As an example, if we compare the performance of the robots following path number 3, we realise that the gap between the rectangular obstacle and the wall is narrow enough to involve some execution problems that have been solved using reactivity. Obviously, we have also counter-examples, as the paths number 4 and 5. Actually, path 4 avoids a non-existing piece of wall that causes an unnecessary displacement. The worse case of avoiding a non-existing wall is when the robot follows the fifth path from the extended map. This path turns around the corner until it reaches an area previously visited by another robot. Although this tendency of going over previous robot trajectories can be a disadvantage, already explored free space also mean safer paths.

Finally, when extension is correct, resulting paths are more informed, and therefore, reactivity is less often needed (see the performance of path number 6). Reactivity is less efficient and can yield the problem that appears path 7 from the non-expanded map. In this case, the reactivity makes the robot to take so many wrong decisions that the robot stops far before reaching the goal. (As the path following section explains, the robot position error grows so much that the error rectangle already includes the goal).

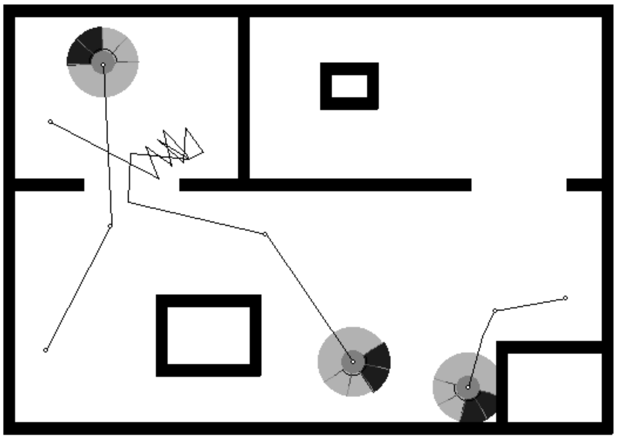


Figure 8.17: Resulting trajectories when trying to follow the planned paths to goal positions 1, 4, and 5 on the non-extended map.

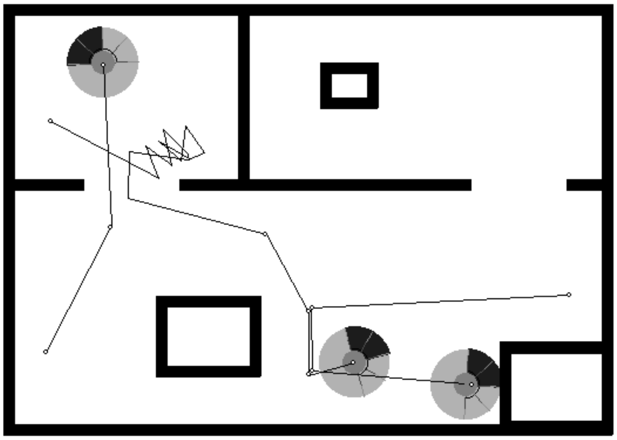


Figure 8.18: Resulting trajectories when trying to follow the planned paths to goal positions 1, 4, and 5 on the extended map.

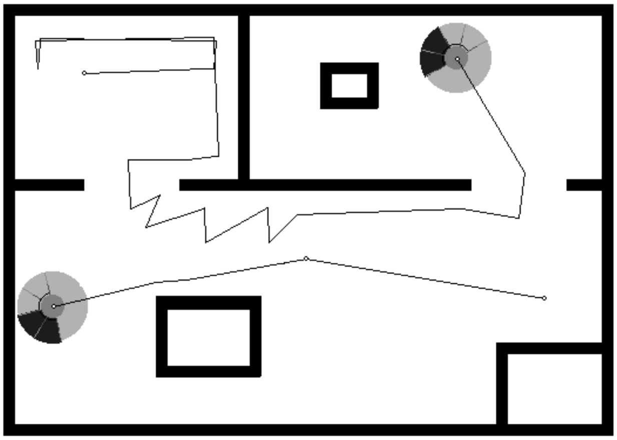


Figure 8.19: Resulting trajectories when trying to follow the planned paths to goal positions 2 and 6 on the non-extended map.

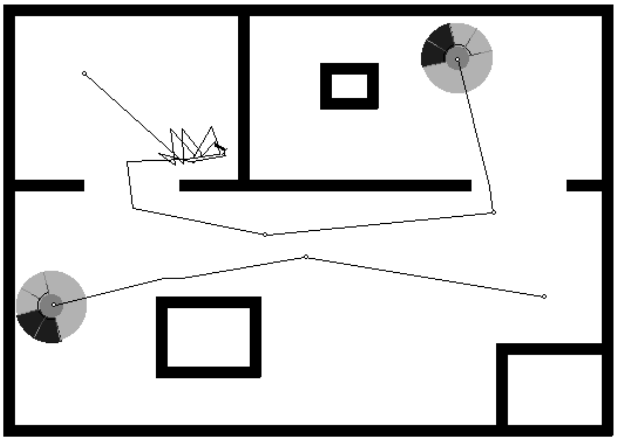


Figure 8.20: Resulting trajectories when trying to follow the planned paths to goal positions 2 and 6 on the extended map.

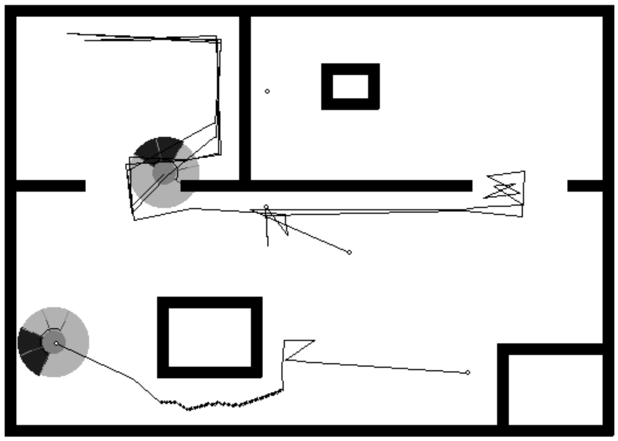


Figure 8.21: Resulting trajectories when trying to follow the planned paths to goal positions 3 and 7 on the non-extended map.

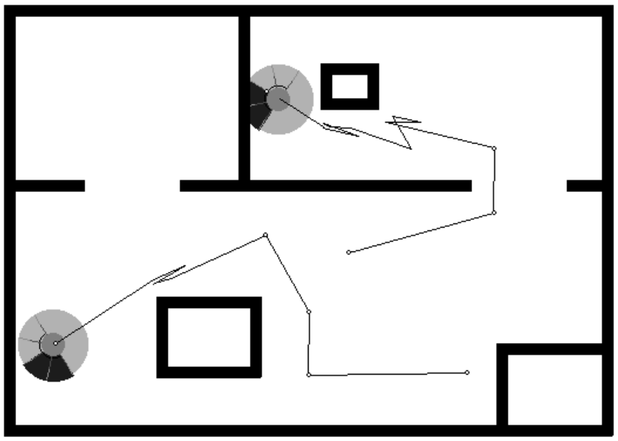


Figure 8.22: Resulting trajectories when trying to follow the planned paths to goal positions 3 and 7 on the extended map.

Part III:Using Symbolic Grouping for Outdoor Environments

Chapter 9

Mapping an Outdoor Environment for Path Planning

The work presented in this third part was developed during my stay as visiting scholar at the Robotics Research Laboratory of the University of Southern California (USC), USA. Under the supervision of Prof. George A. Bekey, I joined the Taskable Heterogeneous Robot Colonies project[3]. The generic goal of this project is to achieve outdoor co-operative tasks with a group of heterogeneous ground based[4] and airborne vehicles[5]. In the work presented herein, an incremental map building approach is applied by heterogeneous vehicles, which accomplish co-operation by sharing information about the environment. Environment information comes into the mapping process from two different sources: aerial images from a helicopter and sonar readings from several ground robots. Ground robots use the resulting map to plan paths towards goal positions. These paths avoid detected obstacles and are updated when there is new information about an obstacle obstructing them.

We consider environment uncertainty depending on the reliability of the information. In the previous parts, the wall following strategy implied that

[3] The Taskable Heterogeneous Robot Colonies project (http://robotics.usc.edu/projects.html), is a project supported by the Defense Advanced Research Projects Agency (DARPA: http://www.darpa.mil).

[4] Ground robots are Pioneers. Pioneer information can be found at the following URL: htpp://www.activmedia.com/robots

[5] AVATAR is a helicopter (Autonomous Flying Vehicle) described at: http://www-robotics.usc.edu/robots/brochure/avatar_bergen.html

the main source of uncertainty was robots' odometry errors. In the present project, we use uncertainty as an estimation of the real existence of detected obstacles and we apply it in order to plan paths that may need to go trough non-real obstacles.

9.1 Project Description

The global scenario of this project is a group of autonomous vehicles that consist of a helicopter and five ground robots that cover a certain area of an arena or reach a specific position specified as a command from a human. Our work has been focused in the representation of the environment as well as in the generation of paths that can be useful for the ground robots to avoid obstacles while reaching the goal position. Considering all the obstacle information that is available at each time, we obtain the shortest paths. However, we do not always use all the information details, and therefore, we can not guarantee optimality (in terms of distance) although we obtain optimal paths for each level of detail.

The utility of a path strongly depends on the reliability of the map information. In our approach, we have two different sources of information: first, the helicopter has a camera facing the ground that provides a birds-eye view of the arena, and second, each ground robot has seven sonar sensors that detect ground obstacles. Robots communicate the obstacle information they gather in order to complete the map of the environment. (The communication is done through a wireless LAN). Since every robot receives information about the obstacles detected by the rest of the robots, all of them have the same individual maps (or similar, in case of transmission problems). In this manner, the distribution of information among the robots allows other robots to use information gathered by a robot that reached a specific region before them. Nevertheless, this does not prevent any robot to plan its own path under bad communication circumstances: since a robot keeps its own version of the map it can still plan a path albeit with less information.

In the following subsection we describe how information from the different sources is pre-processed and used as input data for mapping. The second section gives the details of how our incremental mapping approach is based on the grouping of environment obstacles. The resulting map is used to generate paths, and the third section illustrates this planning process. Finally, section number four explains the way in which we include uncertainty into the mapping process.

9.2 Obstacle Information Extraction

As we have already said, two different kinds of robots gather and share information from the environment. On the one hand, the helicopter captures aerial images and, on the other hand, ground robots detect obstacles by means of their sonar readings.

During the helicopter's flight, image processing will be performed by its exclusively dedicated image system. Next Figure 9.1 shows an image of a typical outdoor environment, a parking lot. This image is of the same kind as the ones captured by the helicopter camera. Since the helicopter chassis development is still in progress, this image was taken from a building (the perspective is the main difference between the neighbouring images that the helicopter will obtain and this image) and processed on a sun station. Image processing has been done using Matlab 5.2 and gives as output a set of polygons that contain some of the obstacles in the real world. The reason for using polygons as descriptions of environmental features is that they constitute a simple and compact way of representing information, so that their communication still allows the robots to perform other tasks simultaneously.

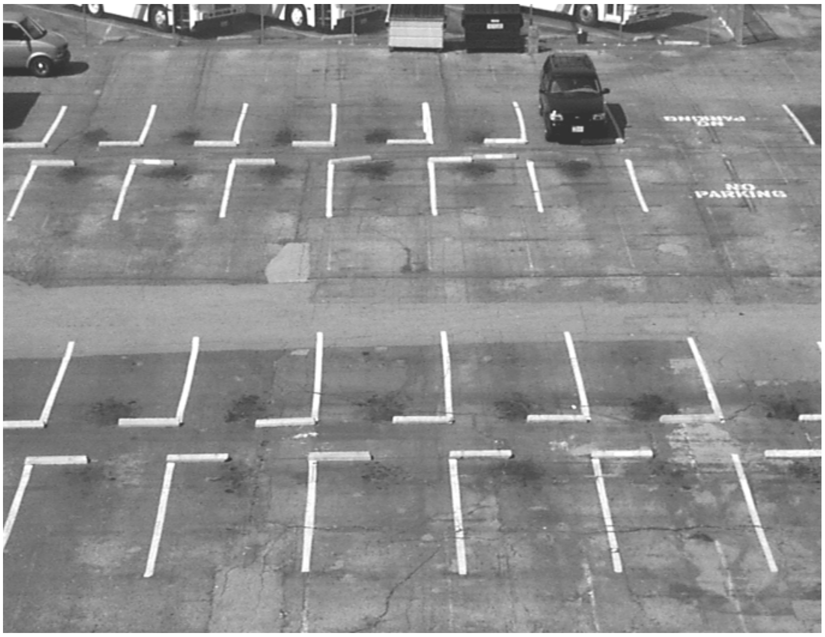


Figure 9.1: Outdoors environment image.

The following steps describe how to extract polygons from a grey-scale image (Figure 9.2 shows the results of applying these steps to the previous image in Figure 9.1):

1. Obtain the most common grey that appears in the image and consider it as background colour.
2. Define lower and upper thresholds so that they specify the range of greys around the background colour that are still considered as background.
3. Transform the grey image into a binary bitmap setting the background pixels to 0 and the remaining ones to 1.
4. Apply several image processing Matlab functions (erode, dilate, clean and majority) to increase the quality of the image by removing spurious pixels.
5. Identify the areas that result from the grouping of neighbour white pixels.
6. For each of the resulting areas, compute its Convex Hull, which is the smallest convex polygon that includes an area. Polygons are specified as a list of vertices. Some of these vertices can be discarded without causing a significant change in the area of the polygon. (As an example, Figure 9.2 had 60 vertices per polygon on average and was reduced to an average of 20 vertices per polygon).
7. Compute the area of each polygon and eliminate those having an area smaller than a certain threshold.

Essentially, this sequence of pre-processing steps separates the foreground from the background and attempts to represent all foreground entities as polygons. Since the helicopter has no stereo vision, the image processing system cannot distinguish real obstacles (in the sense of ground protuberances) from shadows or paintings, therefore there is no guarantee that a polygon will indeed define an obstacle in the environment. See, for example, the painted vertical parking lines which are considered to belong to the horizontal white obstacles in Figure 9.2. Although obtained polygons do not thus necessarily correspond to real obstacles (because they may represent part or groups of obstacles), they can still be considered as being helpful in identifying areas of potential danger for the ground robots.

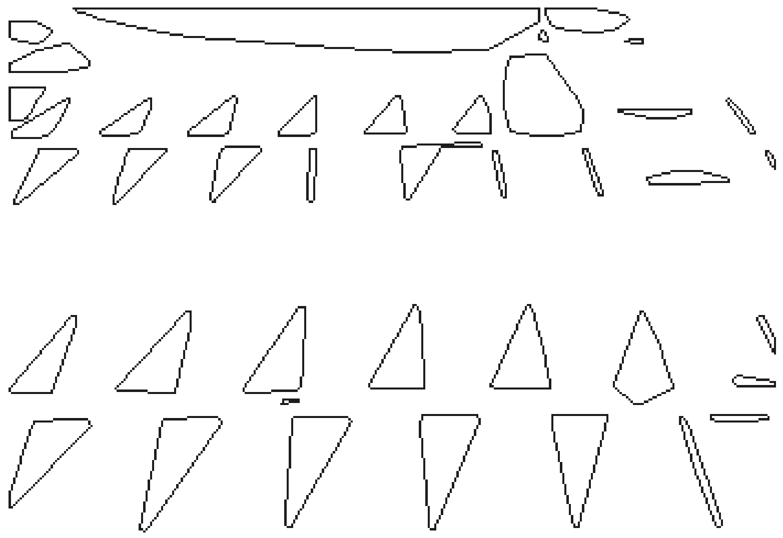


Figure 9.2: Resulting polygons from the previous figure image processing.

The information obtained from the image processing system is incomplete, so the ground robots will need to apply reactive techniques to deal with those obstacles that where not identified in the image but are detected by sonar. This obstacle information is also added to the environment representation and shared among robots. In order to keep a homogeneous representation of the obstacle information that comes from different sources, a robot that is avoiding an obstacle can consider consecutive sonar readings to generate a polygon that approximates the edge of the obstacle. The following Figure 9.3 illustrates an example of how a robot follows part of the edge of wall and the resulting polygonal approximation. Basically the polygon comes from the grouping of consecutive readings that are approximated by a line, so resulting polygons are in fact rectangles containing those lines, with different length and orientation, but with a fixed width that has been defined by default. In fact sonar readings are filtered in such a way that distant readings and groupings that yield segment lines shorter than a threshold are not considered.

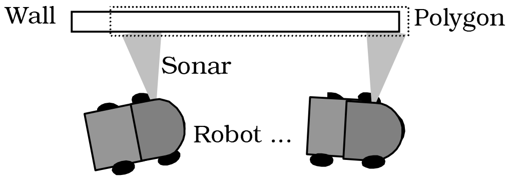


Figure 9.3: Ground Robot Obstacle detection

Although the presented work is focused in the image information extraction, just comment that simultaneously to this work, I was collaborating with Gogksel Dedeoglu[6] for grouping sonar readings linearly. It can be applied when the same wall or obstacle edge is detected more than once. Two thresholds (angle and distance) define when two linear groups of readings are considered to come from the same obstacle. And, when this is the case, comparing the orientation of the first detection with the most recent one shows how the orientation error has increased through time. Therefore, the latest segment detection can be rotated to match the same orientation as the first detected segment, and this can be applied to subsequent detections. Once two segments are parallel, the distance between them is used to correct the position of the robot (it is translated in the perpendicular direction of the reading).

9.3 Map Representation

The previous section defined how obstacle information has been retrieved and specified through polygons, which constitute the input data of the mapping task. This section describes how these polygons are used to incrementally create a map of the environment. Since each robot has a map, every robot updates its own map whether it receives a polygon detected by another robot or it detects the obstacle by itself (and then broadcasts it as well). Thus, all robots apply the method described here simultaneously.

When choosing a representation of outdoor environmental features, there are several characteristics we must take into account. On the one hand, outdoor environments are especially difficult to map because the shape and distribution of obstacles are less predictable than human-made obstacles in indoor environments. This makes it difficult to identify and describe obstacles, particularly, when the information comes from a single camera and local sensors embedded in robots with dead-reckoning errors. On the other hand, it is important to deal with the uncertainty associated with each map element when choosing the right map representation. In the second part of this thesis, we have used a grid representation to model indoor environments. Nevertheless, it is not flexible enough in terms of obstacle positions or contradictory sensor information when used for outdoors environments. The reason is that, for example, if an obstacle that corresponds to more than one cell is represented in the grid and afterwards new information suggests that the obstacle representation should be associated to other cells in the neighbourhood, there is not a direct way of moving it.

[6] Ph.D. student at USC: http://www-robotics.usc.edu/~gogksel

Furthermore, if we want to maintain resolution, then the representation size increases with the size of the environment. This is not the case for a symbolic representation of the environment such as a graph with obstacle areas as nodes and their relations as edges (see for example the works of Kortenkamp 93, Prescott 96, Levitt 90). In such a graph representation, location is just a characteristic —that in some cases can even be reduced to be implicit in node relations— and size depends on the number of obstacle areas, which keep the problem tractable when outdoors.

It is also crucial to have a task oriented representation, that is, to represent information in such a way that it is easy for the robot to use it for performing its task. In our case, the ground robots will use the map for planning paths that go from their initial positions to the goal. We are already familiar with graph representations and they have been proved to be suitable for optimal path planning. And this constitutes a reason, that added to the previous ones listed above, to choose a graph representation for outdoors mapping. In particular, our map consists of a Visibility graph (see Sect. 8.2.1).

9.3.1 Grouping Obstacle Information into the Map

Considering the polygonal information that the robots gather from the environment, the process of building a Visibility graph is direct although the visibility check for all node pairs —i.e. polygon vertex pairs— may be unnecessarily expensive in terms of computational time. The purpose of our mapping task is not to give a detailed description of an environment, but a rough approximation of those environmental features that can be a potential danger for the ground robots when approaching a target position. What we propose is to build an efficient higher level abstraction on top of the polygons such that a robot is able to plan its path towards a goal position at this level. Nevertheless we keep the relation between both levels of representation so that if occasionally the robot needs a more accurate path, it can go down into the polygon level and still use the abstract level to simplify the amount of treated data. In general, this should not be the case because outdoors environments usually have a relatively low obstacle density. This makes unnecessary for the robots to know the exact shape of an obstacle in order to avoid it.

We characterise the higher level of abstraction as a set of *Obstacle Areas*. An obstacle area being a rectangle that includes a group of overlapping polygons. Polygons can intersect due to different reasons: on the one hand, the vision system can provide polygons that overlap, and on the other hand, an obstacle —or part of it— can be detected several times, by different means or from distinct perspectives. Thus, map information is distributed over two levels: the first one defines how the obstacles in the environment are distributed in different areas, and the second level goes into more detail and specifies the polygons that belong to each area. Figure 9.4 presents another example of an outdoors image and Figure 9.5 shows how their corresponding image processing polygons have been grouped into rectangular obstacle areas.

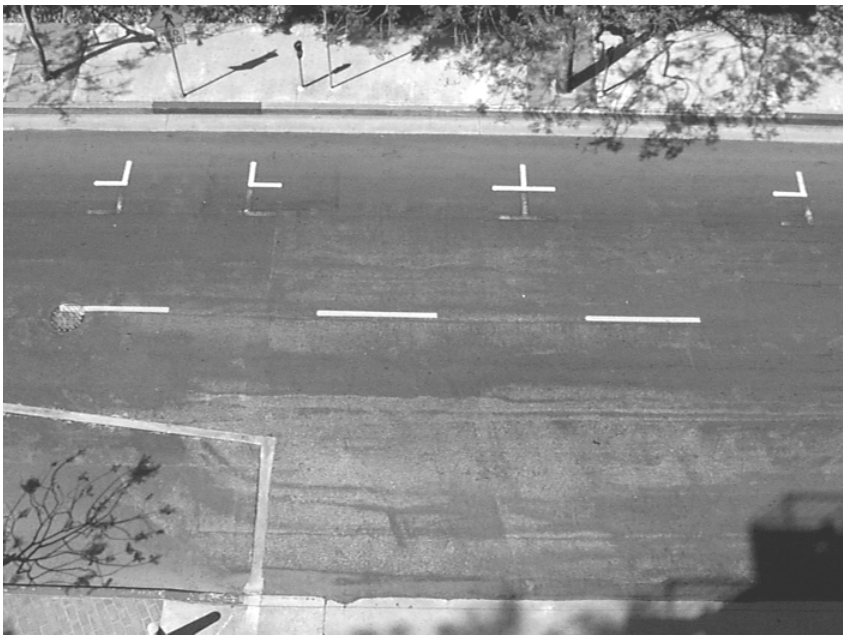


Figure 9.4: Outdoors environment image.

Figure 9.5 gives a good description of the kind of obstacle areas that we can find. Isolated obstacles in the environment yield obstacle areas with a single polygon, in these cases the gain comes from the fact that we reduce an average number of twenty vertices per polygon to a constant number of four. This might not seem a big improvement in the performance, but it has the advantage that it does not dismiss important information. Since it is an isolated obstacle the robot will still have room to surround the obstacle area rectangle.

Bigger obstacle areas are usually composed by several overlapping obstacles. In these cases, they tend to cover bigger free space areas —which increase the risk of covering useful areas for path planning. However the reduction of the number of vertices grows in proportion to the number of included polygons. Nevertheless, the vertices of those polygons that are completely contained by others are discarded without taking any additional risk.

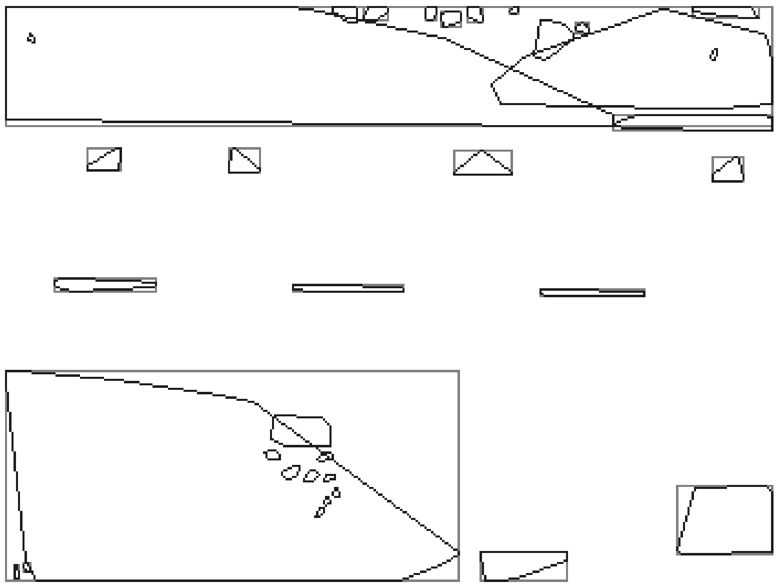


Figure 9.5: Grouping obstacles: obstacle areas are defined as rectangles containing intersection polygons.

Finally, there is a special kind of obstacle areas that are produced when polygons do not intersect but the rectangles of their corresponding obstacle areas intersect. We call them *intersecting obstacle areas*. Usually they require the system to go down to the polygon level to treat the relations among them. Intersecting obstacle areas are the ones that provide less efficiency gain because going to the polygon level means getting closer to the original computational costs, but in the following sections we will show that we still have some gain in such cases.

9.3.2 Map Structure

Our map representation consists of a set of obstacle areas and a visibility graph:

- *Obstacle areas* present two levels of information: the highest level specifies a rectangle that contains a set of intersecting polygons, and the lowest level includes the list of these intersecting polygons. In case an obstacle area is intersecting with others, the list of references to these areas is also stored in the high level. For the low level, it can also compute an additional polygon representing the union of the polygons in the obstacle area. Union polygons are computed under request when refining the path that has been obtained in the high level representation; we will see how they are useful for visibility computations.

- We create the *visibility graph* by considering as nodes the vertices of the obstacle area rectangles and by establishing edges between 'visible' nodes: those node pairs without any obstacle in between. Edges are labelled with the Euclidean distance among the nodes they relate. In our approach, visibility is computed considering the obstacle area rectangles in stead of the polygons.

Figure 9.6 illustrates how we represent the map information. Figure 9.6 a) shows a subset of three obstacle areas from the nineteen obstacle areas in the previous Figure 9.5. Each area has been drawn as its rectangle and the polygons it groups, (for example, obstacle area number 1 groups six obstacles).

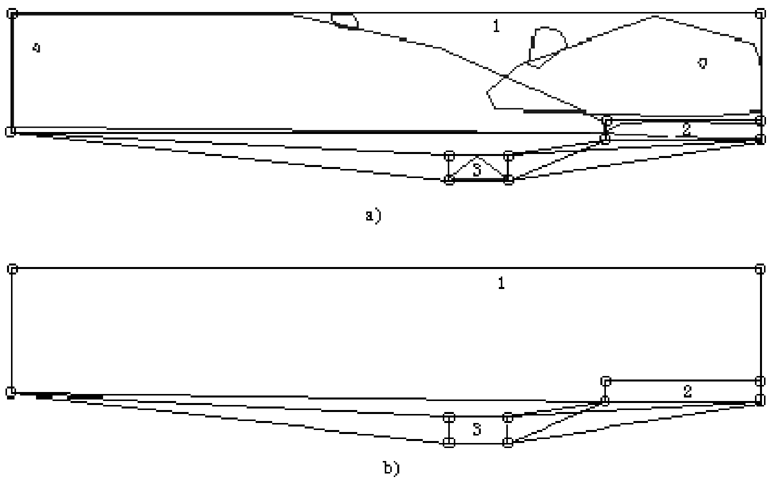


Figure 9.6: a) Subset of obstacle areas from the previous Figure 9.5. b) Its corresponding visibility graph.

Although it has not been shown in the figure, obstacle areas number 1 and 2 have a cross-reference between them that states their intersection. Figure 9.6 b) is the corresponding visibility graph. Small circles represent nodes in the graph and come from the vertices of the obstacle area rectangles. However, not all the vertices are included because there are vertices in the intersecting areas that may lie inside an obstacle of one of the areas it intersects with. This is the case of the lower-right vertex of the rectangle in area 1, which is located inside the polygon of the second area.

Keeping the level of abstraction at the obstacle-area-rectangle level speeds up the computation of the visibility. The gain depends on the number of polygons that each obstacle area is grouping and the number of vertices of each polygon. But the gain can be significant if we take into account the fact that all node pairs check for every obstacle area if the area obstructs their visibility.

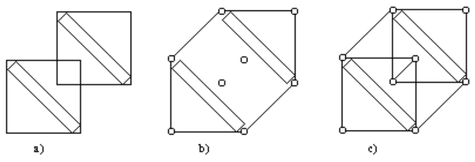


Figure 9.7: a) Intersecting obstacle areas, b) checking visibility between nodes considering obstacle area rectangles, c) visibility considering polygons.

Unfortunately, the high level of abstraction may yield a low connection between nodes that belong to intersecting areas. This is not the case of the intersecting areas in the previous Figure 9.6 but for a case like the one shown in Figure 9.7, it is necessary to check visibility at the polygon level because the higher level does not connect nodes between the two parallel obstacles.

9.3.3 Map Updating

After determining how data is represented, we describe how robots share information and how they include new information into the current map. The idea is that each time a robot defines a new polygon it broadcasts it to the rest of the team. Thus, all robots are able to update their maps. To add a new polygon into the map means that it must be included into one of the obstacle areas and that the visibility graph must be updated. Graph update is computationally expensive, because it implies including new nodes to the graph and checking if the new polygon obstructs the visibility of those pairs of nodes that were visible before the new polygon is taken into account. However, this process should be kept as cheap as possible because there might be some information that would turn out to be useless for some robots —that is, information about obstacles that are located far away from one specific robot trajectory. We accomplish that by considering again high level information. The following pseudocode depicts the updating algorithm and in the rest of this subsection we will see how Update uses the high level information (i.e., obstacle area rectangles) to restrict the computation.

Let Map be a list of obstacle areas $OA_{i\ (i=0..n)}$, where each obstacle area contains a list of polygons $LP_i = \{P_1,..P_k\}$ and let G be the visibility graph associated to Map. Then we apply the Update function to include a new polygon P' into Map.

```
Update (Map, P')
{  list_of_intersecting_areas = ∅
   last_area_with_polygon = ∅
   polygon_added = no
   Repeat for all OAi obstacle areas ∈ Map
   {  intersection = Check_Intersection(P', OAi)
      If (intersection = intersects_a_polygon_of_LP)
      {  If (last_area_with_polygon = ∅)
            Include_polygon_in_area (P', OAi)
         Else
            Fuse_areas (last_area_with_polygon, OAi)
         last_area_with_polygon = OAi
         polygon_added = yes
      }
      Else
        If (intersection = only_intersects_the_rectangle)
           list_of_intersecting_areas = OAi
   }
   If (polygon_added = no)
   {  OA' = Create_new_Obstacle_Area (P', list_of_intersecting_areas)
      Add_Obstacle_Area (Map, OA')
   }
}
```

Initially, the Update function checks if the new polygon P' intersects with any of the obstacle areas OA_i. An increase in efficiency comes from the fact that when P' does not intersect with the rectangle that is defined for an area OA_i, there is no need of checking the intersection for the polygons inside ($LP_i \in OA_i$). When P' intersects the rectangle and it overlaps polygons of LP_i, then LP_i must contain P': if OA_i is the first area that intersects P', then P' is added to LP_i, otherwise it means that P' has already been included in another $OA_{j,\,j\neq i}$ and both areas must be fused in order to have all overlapping polygons grouped under the same area $OA_j' = OA_j \cup OA_i$ (with $LP_j' = LP_j \cup LP_i$). Another advantage of the high level information appears when a polygon is included into an area OA_i in such a way that its rectangle does not grow. In this case, the associated graph G that corresponds to the previous Map is still valid and a review of its visibility is not needed (although G may need it when OA_i is an intersecting area).

Information about P' intersecting the rectangle of OA_i without overlapping polygons of LP_i, causes the reference of OA_i to be stored in the variable list_of_intersecting_areas. In that way, when P' will be added to another area $OA_{k,}$ $_{k\neq i}$ all areas (i.e., OA_k together with the ones in list_of_intersecting_areas) will become intersecting areas. Finally, if after checking the intersection of P' with all obstacle areas OA_i, P' has not yet being included, a new obstacle area will be created and included into the Map. Notice that, in fact, this algorithm can be used to update empty maps because it simply creates a new obstacle area for the first polygon (and updates G correspondingly). Hence, this is the function that we use to incrementally create our map.

9.4 Path Planning

When using a graph, the shortest path between two positions is given as a list of connected nodes in the graph so that it is possible for the robot to go from the initial point to the goal following the links between the positions that the path nodes represent. In general, the initial position of a robot and its goal are not represented as nodes in the graph. Therefore, to compute the path between two positions, it is necessary to first include them as nodes into the visibility graph (and, of course, establish their connections by computing their visibility). Then, the A* algorithm is applied to obtain the path. A* is optimal for this problem because we can use the Euclidean distance as a heuristic and its triangular inequality property makes it conservative (see Sect. 8.2.2).

When the shapes of obstacles are well known and can be approximated by polygons, visibility graphs yield optimal paths. Unfortunately, we cannot guarantee optimality because of the robots' limited obstacle sensing, but we can come as close as the information allows us to be. If paths that are close to obstacles present a danger, then it is always possible to grow the obstacles by half the width of the robot, to provide some margin.

Obstacle expansion is also useful to guarantee paths that can be actually traversed by the robot. This step can be done in the data acquisition stage, however, we have not done it because we deal with polygons that do not correspond to real obstacles and growing them can only yield to poorer paths. Our underlying idea is, therefore, to have robots that are able to compute an approximated path towards a goal position, and then a robot can roughly follow the path, applying reactive techniques when its internal map does not correspond with the environment that it is actually sensing.

The following Figure 9.8 shows an example of the kind of paths that result when applying the planning algorithm to the visibility graph G. In this case, we consider the obstacle areas —defined for the polygons obtained from Figure 9.1— that are shown in Figure 9.2. Taking the upper left corner to be the origin of the images and considering pixels as image units, a path between the initial and final position was asked to be planned.

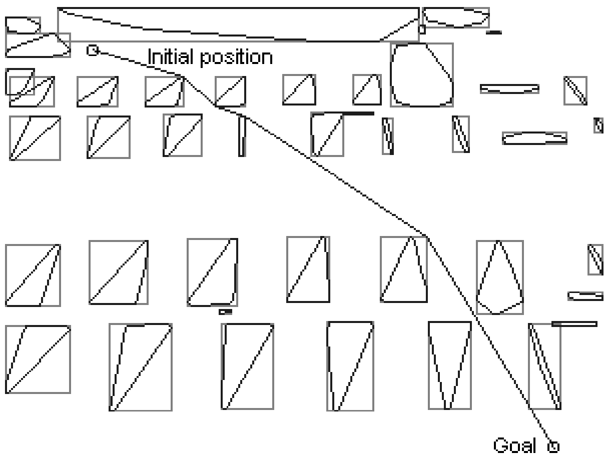


Figure 9.8: Path that considers high level information in the map from image in Figure 9.1.

The initial and final point co-ordinates are (100, 50) and (640, 520) respectively. Since G has been created considering the higher level of information, the resulting paths go trough the vertices of the obstacle area rectangles. In this manner, the resulting path is specified by the following sequence of positions:

(100, 50), (207, 82), (245, 117), (279, 129), (492, 272), (640, 520).

9.4.1 Path Update

Adding information into the map does not necessarily mean changing the current path. In fact, a robot can be following its own path and receiving simultaneously information about other locations in the environment without changing its trajectory. Only information about obstacles that are obstructing the current path triggers the path planning algorithm. Figure 9.9 illustrates how the path is updated when the information in Figure 9.2 is partially received. The image a) of Figure 9.9 shows the path generated by a ground robot that is located at the initial position and has only received twenty polygons from the helicopter image processing system. The path in image b) has been generated after receiving ten more polygons from the helicopter. The corresponding paths have been specified as:

Path a): (100, 50), (130, 83), (183, 177), (640, 520).

Path b): (100, 50), (164, 117), (229,128), (272, 177), (387, 272), (438, 347) (640,520).

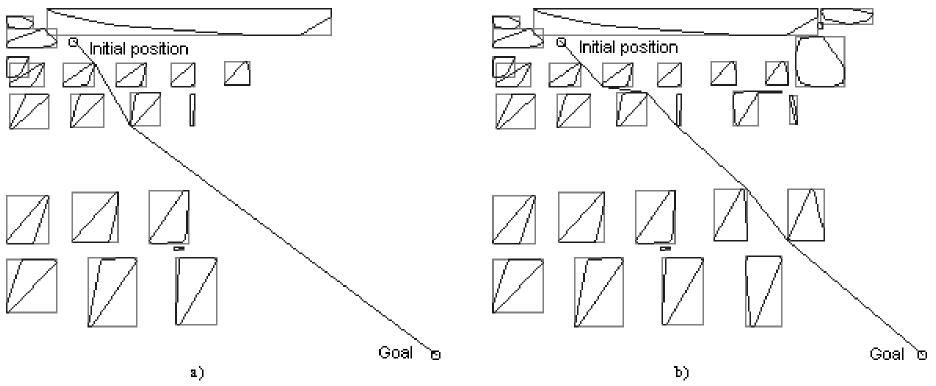


Figure 9.9: a) Path planning considering twenty obstacles; b) The path must be redefined after the addition of ten new polygons that obstruct the previous path.

9.4.2 Refining the Path

In the previous sections we have seen how we can consider the higher level information and still obtain useful paths. Nevertheless, when the image processing provides obstacle polygons that are a rough approximation of the real obstacles and the resulting obstacle areas cover most of the space in the environment, it may be worth to work at a lower level information without losing more accuracy. The following Figure 9.10 is an example of such situation, and the extracted obstacles together with the resulting obstacle area rectangles are shown in Figure 9.11. In this figure, the path obtained considering the high level information has been displayed together with the path that uses the lower level of detail. They have been labelled *path* and *repath* respectively. As expected, the use of more accurate information results in a refined path that is shorter than the higher level path. In fact, this is always the case and the gain depends on how accurate the rectangle areas approximate the obstacles they contain (obviously, the gain is 0 for those paths that avoid obstacles going trough obstacle vertices that coincide with the rectangle vertices).

In the figure, the sequence of positions in the resulting paths are:

Path: (5, 100), (113, 87), (400, 500)

Repath: (5, 100), (159, 193), (163, 198), (400, 500)

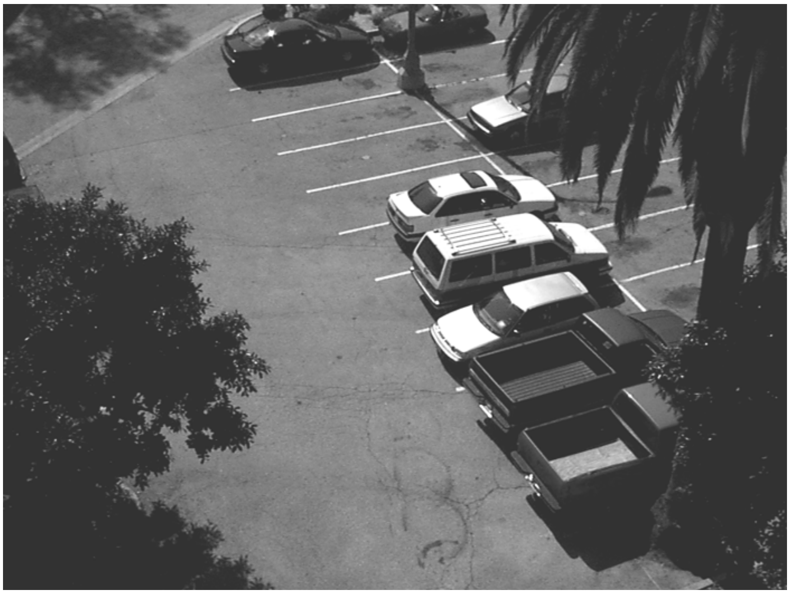


Figure 9.10: Example of an image that might need a path refinement.

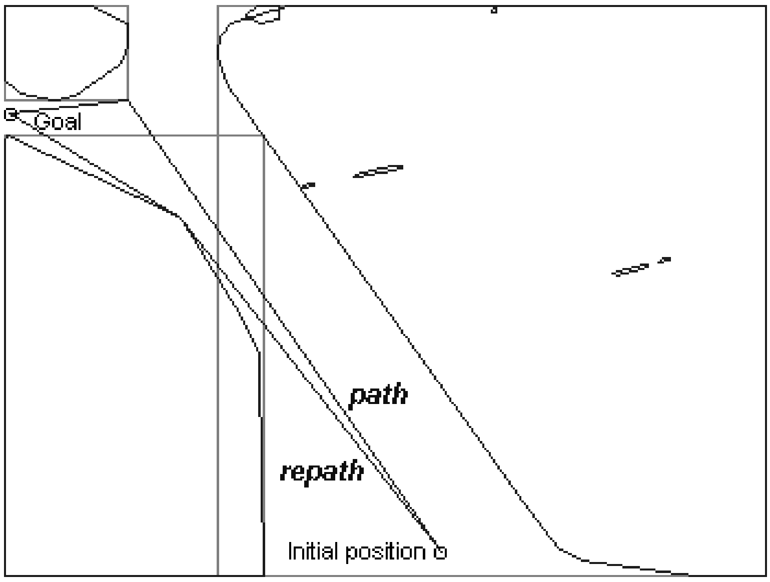


Figure 9.11: Path planning at two different levels: *path* considers obstacle areas whilst *repath* considers obstacle polygons.

Note that the two intermediate points in *repath* are very close due to the polygon shape. The fact that these two points are closer to the image representation of the obstacle is not necessarily dangerous. In this case, for example, real obstacles are included into the polygons far from their boundaries, and in addition to that, the polygon that is reached by repath does not correspond to a ground obstacle but to the top of a tree on the left.

When going from the high level information down to the lower level, what we are doing is to generate a new visibility graph that has as nodes the vertices of the obstacles instead of using the vertices of the obstacle area rectangles. The next Figure 9.12 shows an extreme case that illustrates the difference. In a) two obstacles are included into their respective obstacle area rectangles. The problem is that some of the rectangle vertices lay inside the obstacles, and therefore, they are not included as nodes into the visibility graph. Therefore, the visibility graph that results from considering the high level (see part b)) cannot generate any path going between the two obstacles. On the contrary, Figure 9.12 c) shows how this is possible when the low level information obstacle vertices are used.

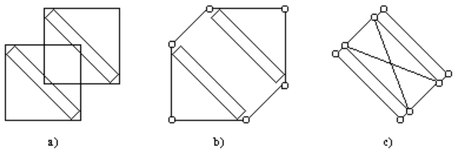


Figure 9.12: a) Two obstacle polygons and their corresponding obstacle area rectangles. b) The resulting high-level visibility graph. c) Visibility graph at the low level.

Planning at the polygon level of detail is a computationally expensive process that means to create a new graph including all polygon vertices as nodes and computing the visibility graph between them. Fortunately, we can still use the path obtained from the high level planning to choose the obstacles on which lower level planning should be focused.

The basic idea is that information from a less accurate path can help the planning process to discard polygons that are far away from the path. Therefore our approach is to always compute the plan at the higher level, and if required, use it to create a local lower level map in which to apply planning. This local map is built using the obstacle areas that contain the nodes in the path. The difference now is that instead of including as nodes the vertices from the obstacle area rectangles, we include the vertices of the union polygon. The union polygon is an optional part of the obstacle area that is now computed by combining all the polygons inside the area. For those areas having more than one polygon, this can reduce the number of vertices to include into the graph, especially for polygons contained in others.

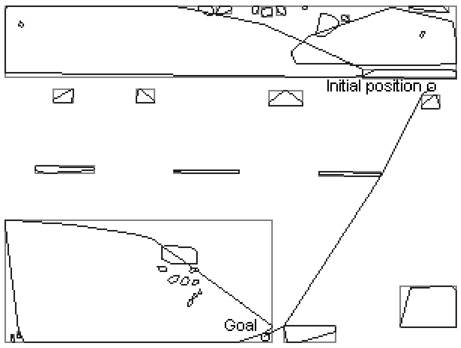


Figure 9.13: Planned path in the map from Figure 9.5.

As an additional example, we can use the map from Figure 9.5 to plan a path between an initial and a goal position. Figure 9.13 above, shows the first path that has been computed considering the higher level of information. In that manner, we use this path to choose the four obstacle areas that are used to build the local map that appears in the following Figure 9.14. In this case, the refined path at part a) is very similar to the high level path due to the accuracy of the rectangles in approximating the obstacles of the involved obstacle areas. We have chosen this example to illustrate the amount of obstacle areas that can be discarded when building the low-level visibility graph (which appears at Figure 9.14b)). Figure 9.14 a) contains the union of the polygons of the involved areas and we can observe how the refined path goes closer to the obstacles than the one in the previous Figure 9.13.

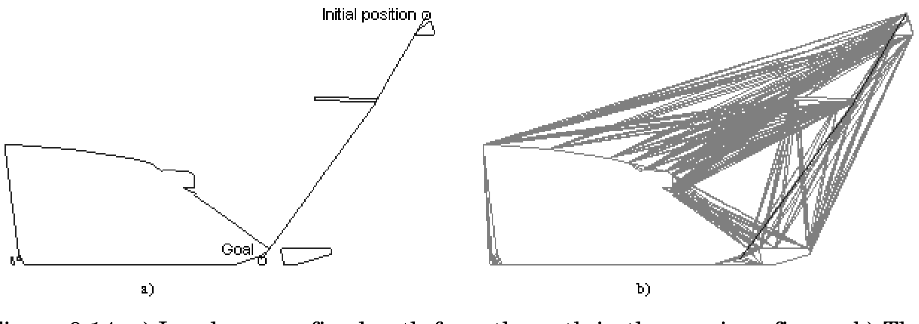


Figure 9.14: a) Local map: refined path from the path in the previous figure. b) The corresponding visibility graph.

It is not always the case that the refined path that results from the planing on the local map avoids all the obstacles in the global map. In order to guarantee that, it is necessary to go up to the obstacle area level and verify that there are no obstructing global map areas. In this manner, if there is any obstructing area, it is included in the local map and the local planning process is repeated.

In addition to the refinement of paths, both Figure 9.13 and Figure 9.14 illustrate another especial situation of the path planning. In cases where the rectangles of the obstacle areas cover bigger areas than the obstacles they contain, it is possible that the initial or final points of the path lay inside an obstacle-area rectangle without being located inside any obstacle of this area. As we have already said, the corresponding point must be included into the visibility graph establishing the relations with the rest of the nodes in the graph. The difference now is that we compute the visibility by using a combination of both levels of information. On the one side, we consider the obstacles inside the obstacle area that contains the point and, on the other side, we still use the high level information —i.e., the rectangles— for the rest of the obstacle areas.

9.4.3 Considering Uncertainty

When building the map of an environment, there can be different sources of uncertainty. As we have seen in the previous parts of this thesis, odometry and sensor errors are the main cause of uncertainty in obstacle positioning. However, if we consider that in this outdoor approach obstacle information is mainly extracted from an aerial camera, uncertainty about what is and what is not an obstacle becomes the main concern. Shadows, different materials or colour changes are just some of the environment features that can lead the vision system to the identification of wrong objects.

One way of facing this uncertainty is to associate with each polygon a certainty degree of its correspondence to real obstacles in the environment. We obtain the certainty degree by taking the product of the area of the polygon and the reliability of the sensor that detected the polygon. In our case we have two kinds of sensors —a camera and sonar sensors— and we assume reliability as being less than 1 for both sensors. In the same manner, we compute the certainty degree of an obstacle area by adding the certainty degrees of all polygons inside this area.

The addition of uncertainty about the existence of an obstacle changes the concept of the visibility graph. Considering that a polygon might not be a real obstacle, those node pairs that were not connected because of this polygon visibility obstruction, should now have an edge connecting them. When planing over a regular visibility graph, the cost of an edge is usually considered to be the Euclidean distance between the two nodes. Taking now into account node uncertainty, more edges must be added in order to represent relations between nodes whose connections might not be obstructed even if the map has some polygons obstructing them. In this manner, the cost for these edges must be now weighted distances, with the weights depending on the certainty of the obstacles they go through. The value that we use as an edge cost is the Euclidean distance plus the addition of all the certainty degree of those obstacle areas obstructing the visibility of the edge. Thus, visibility is now checked for the complete graph and edge costs are updated by adding or subtracting certainty degrees.

The new approach that results from considering obstacle uncertainty generates paths that can go through low certainty areas when there is not a better path to follow. Obviously, when the path assumes that an obstacle can be passed through and it turns out not to be the case, the robot that is following the path will need to apply reactivity and will probably end up with a longer trajectory. Nevertheless, it is still better to have a path that will hopefully avoid some of the obstacles than nothing (we take this approach when the visibility graph method does not generate any path).

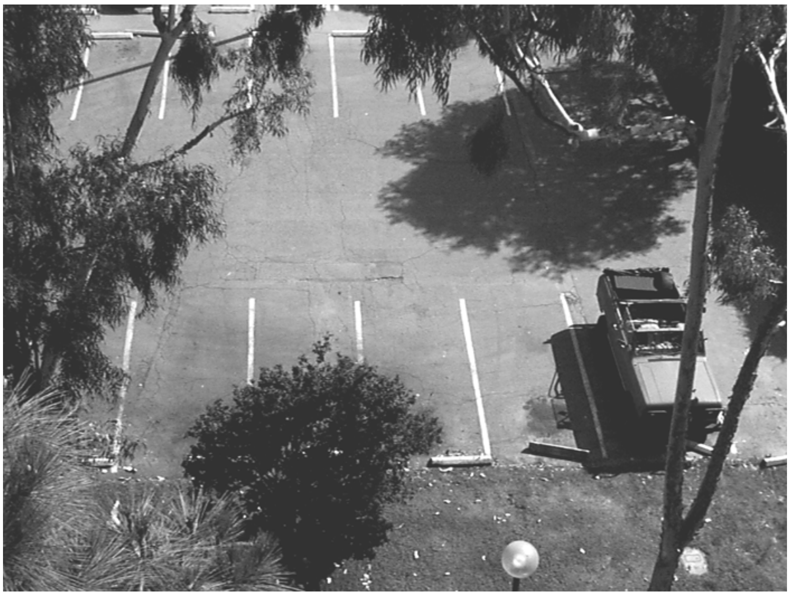


Figure 9.15: Parking image that is partially occluded by a tree.

The Figure 9.15 above shows an example of an image that can generate a map with a high density of non-existing obstacles. In this case, the image processing system produced eighty obstacles, and most of them do not correspond to real obstacles but to shadows, trees, street lights or painted lines. Although this kind of images may seem useless for the ground robots, we can use uncertainty to plan paths going through these non-real obstacles. Since we have no means of distinguishing real from non-real obstacles, path planning is done over a graph where all non-obstructed edges have a much lower cost than those going through obstacles. Therefore, resulting paths avoid all obstacles that are in fact possible to avoid. The path in the following Figure 9.16 illustrates this idea: it goes towards the left side of the parking area at the previous figure. This path ends in a position that is free, although a tree makes it to appear occupied. Notice that going through this obstacle is the only way the path can reach its goal. The rest of the obstacles are avoided so that even if some of them do not correspond to real obstacles, those that do correspond are safely avoided.

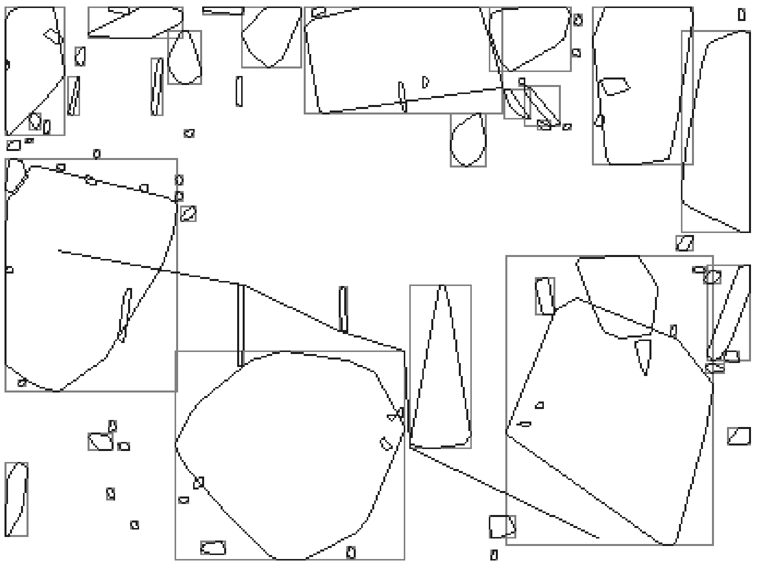


Figure 9.16: Planning with uncertainty: the path goes through an obstacle in the map that does not correspond to a real obstacle.

9.5 Implementation

The application that performs map generation and path planning has been developed in Visual C++ 5.0. It has been designed so that it can be executed by the ground robots (using the image processing information from the helicopter) and therefore it has no graphical interface. In fact, in order to be executed by the Pioneers, the interface of the application had to consider the Pioneer Application Interface (PAI), a simplified and standardised interface to Saphira[7] (Saffiotti et al. 93 and 95, Konolige 98) developed by Barry Werger[8].

Basically, the developed application interface is done through files containing the communicated polygons, the generated obstacle areas, the corresponding visibility graph and the planned path. These files are text files that are displayed using an additional application we developed in order to facilitate the understanding of the results.

Here, we list the contents of each file corresponding to the image in Figure 9.10. Most of these files were displayed in Figure 9.11.

Polygons

This file contains the polygons ordered by regions. First, the number of regions in the file is specified, thus, for each region there is a list with its polygons, which are also specified by the number of points and the list of these points. (We have omitted 5 polygons from the third region with the aim of giving a shorter description).

Number of regions: 3

Region 1: Number of polygons: 1

Polygon 1: Number of points: 17

- points: (30, 1), (2, 1), (1, 11), (1, 69), (2, 70), (11, 78), (15, 81), (45, 87), (48, 87), (66, 83), (106, 55), (107, 54), (113, 37), (113, 18), (84, 3), (82, 2), (78, 1).

Region 2: Number of polygons: 1

Polygon 1: Number of points: 10

- points: (3, 120), (1, 124), (1, 521), (237, 521), (237, 519), (234, 328), (233, 320), (218, 286), (163, 198), (159, 193)

Region 3: Number of polygons: 7

Polygon 1: Number of points: 19

[7] Saphira is a client architecture designed to operate with a robot server: a mobile robot platform that controls the low-level motor operations as well as sensor readings. This server receives commands from the clients and sends them back information. This client/server paradigm abstracts the Saphira client from the particulars of robots and can exist either on the robots itself (as inFlakey or Erratic from SRI International) or off board on a host computer (as in the tiny Khepera, original from the Swiss Federal Institute of Technology). Pioneers are also servers of Saphira that can have it either onboard or connected by a radio modem.
Available documentation about PAI and Saphira can be found at the URL: http://css.activmedia.com/docs

[8] discover him at http://www-robotics.usc.edu/~barry.

- points: (622, 1), (259, 1), (207, 18), (197, 29), (195, 43), (195, 44), (196, 48), (200, 60), (206, 75), (469, 448), (492, 478), (499, 487) (503, 492), (506, 495), (524, 505), (528, 507), (631, 521), (696, 521), (696, 1).

...

Polygon 7: Number of points: 6

- points: (600, 233), (598, 236), (600, 236), (607, 234), (607, 232), (604, 232).

Obstacle areas

The polygons file already contains how they are distributed within the obstacle areas. Therefore, this file only specifies the rectangles defined for the obstacle areas that contain the polygons. By comparing the following file with the previous one, we can observe the simplification gain of using obstacle area rectangles instead of the polygons. In fact, although we use rectangles, any other polygon could have been used, their specification is the same than the rest of polygons: the number of vertices and the list of their co-ordinates.

Number of regions: 3

Region 1: Number of points: 4

- points: (1, 1), (113, 1), (113, 87), (1, 87)

Region 2: Number of points: 4

- points: (1, 120), (237, 120), (237, 521), (1, 521).

Region 3: Number of points: 4

- points: (195, 1), (696, 1), (696, 521), (195, 521).

Graph

The nodes of the visibility graph are defined from the vertices of the obstacle area rectangles. The arcs are then established between pair of nodes. These arcs joining visible nodes are labelled with their Euclidean distance. On the contrary, if an arc relating two nodes is obstructed, then the label includes both the Euclidean distance plus the sum of the certainty of the obstructing polygons (i.e., the areas weighted by 0.5).

Here we list four of the nodes included in the graph text file. For each node, its arcs are classified as being visible or obstructed. Arcs appear as tuples of the form (n): label, being n the node to which the current node is related to, and label the corresponding cost of the arch.

Number of nodes: 12

Node 1: co-ordinates: (1, 1)

- Number of visible nodes = 2: (2): 112, (4): 86.

- Number of obstructed nodes = 9: (3): 4180.7; (5): 4158.5; (6): 4303.8; (7): 39611; (8): 43636; (9): 4233.5; (10): 96394.5; (11): 96567.5; (12): 43671.

Node 2: co-ordinates: (113, 1)

- Number of visible nodes = 4: (1): 112; (3): 86; (6): 171.8; (9): 82.
- Number of obstructed nodes = 7: (4): 4180.7; (5): 4202.9; (7): 43687; (8): 43647.9; (10): 92243; (11): 92441.2; (12): 39602.9.

Node 3: co-ordinates: (113, 87)

- Number of visible nodes = 5: (2): 86; (4): 112; (5): 116.8; (6): 128.3; (9): 118.8.
- Number of obstructed nodes = 6: (1): 4180.7; (7): 39527.9; (8): 39524.7; (10): 92249.3; (11): 92386.8; (12): 39518.2.

...

Node 12: co-ordinates: (195, 521)

- Number of visible nodes = 2: (7): 42, (8): 194.
- Number of obstructed nodes = 9: (1): 43671; (2): 39602.9; (3): 39518.2; (4): 39551.9; (5): 39522; (6): 39479.7; (9): 39596.5; (10): 131458.6; (11): 39577.5;

Path

The specification of the path is as simple as we have seen previously in the 1.1 Path Planning Section. Adding the initial and goal positions to the previous graph, the shortest path is obtained applying the A* algorithm.

Number of points: 3

- points: (5, 100), (113, 87), (400, 500)

Redefined files

When we want to use the lower level information in order to increase accuracy, we use the planned path to select the obstacle areas that will be treated. In the particular case that we are considering, the initial position of the path belongs to the largest area and it goes to the bottom-right vertex of the smallest area before it reaches the goal position. Therefore, these two areas must be included. Nevertheless, since the medium-sized area intersects with the largest area, the three areas are included (that is, there is no variation in the number of considered regions).

Considering the lower level information means to treat polygons inside the obstacle areas instead of its rectangle areas. As we have already said, when this option is used, the system computes the union of all the polygons inside each area. The union is computed by using the Generic Polygon

Clipper library[9] (GPC), a C library developed by Alan Murta[10]. The union does not affect those obstacle areas with only one polygon, but in the case of the third region —which has 7 polygons—, it represents a significant simplification without loosing accuracy.

In this manner, the polygons in the refined map are:

Number of regions: 3

Region 1: Number of polygons: 1

Polygon 1: Number of points: 17

- points: (30, 1), (2, 1), (1, 11), (1, 69), (2, 70), (11, 78), (15, 81), (45, 87), (48, 87), (66, 83), (106, 55), (107, 54), (113, 37), (113, 18), (84, 3), (82, 2), (78, 1).

Region 2: Number of polygons: 1

Polygon 1: Number of points: 10

- points: (3, 120), (1, 124), (1, 521), (237, 521), (237, 519), (234, 328), (233, 320), (218, 286), (163, 198), (159, 193)

Region 3: Number of polygons: 1

Polygon 1: Number of points: 27

- points: (696, 521), (631, 521), (528, 507), (524, 505), (506, 495), (503, 492), (499, 487), (492, 478), (469, 448), (272, 168), (271, 168), (206, 75), (200, 60), (196, 48), (195, 44), (195, 43), (197, 29), (207, 18), (225, 12), (222, 11), (222, 9), (232, 1), (250, 1), (252, 2), (252, 3), (259, 1), (696, 1).

These vertices are now used to build the corresponding visibility graph so that refined paths can be planned. Considering the same initial and goal positions (which are temporally included as nodes in the graph), the resulting path is as follows:

Number of points: 3

- points: (5, 100), (159,193), (163, 198), (400, 500)

There are a total number of 56 nodes in the refined graph, we just list the 4 nodes involved in the refined path. (We specify nodes by means of their co-ordinates).

Node (5,100)

- Number of visible nodes = 14: (1, 69): 31.3; (2, 70): 30.1; (11, 78): 22.8; (15, 81): 21.5; (45, 87): 42.1; (48, 87): 45; (66, 83): 63.3; (3, 120): 20.1; (1, 124): 24.3; (159, 193): 179.9; (271, 168): 274.6; (206, 75): 202.5; (200, 60): 199.1; (196, 48): 198.

[9] GPC is a public software available at <URL>: http://www.cs.man.ac.uk/aig/staff/alan/software/index.html#gpc

[10] Alan Murta works at the Advanced Interface Group of the Computer Science department in the Manchester University (U.K.).

Node (159, 193)

- Number of visible nodes = 24: (11, 78): 187.4; (15, 81): 182.4; (45, 87): 155.7; (48, 87): 153.5; (66, 83): 144; (106, 55): 147.8; (107, 54): 148.4; (113, 37): 162.6; (113, 18): 180.9; (3, 120): 172.2; (163, 198): 6.4; (506, 495): 460; (503, 492): 455.8; (499, 487): 449.5; (492, 478): 438.3; (469, 448): 401.4; (271, 168): 114.8; (206, 75): 127; (200, 60): 139.2; (196, 48): 149.6; (195, 44): 153.3; (195, 43): 154.3; (197, 29): 168.3; (5, 100): 179.9.

Node (163, 198)

- Number of visible nodes = 19: (106, 55): 153.9; (107, 54): 154.5; (113, 37): 168.6; (113, 18): 186.8; (218, 286): 103.8; (159, 193): 6.4; (506, 495): 453.7; (503, 492): 449.5; (499, 487): 443.2; (492, 478): 432; (469, 448): 395.1; (271, 168): 112.1; (206, 75): 130.3; (200, 60): 142.9; (196, 48): 153.6; (195, 44): 157.3; (195, 43): 158. 3; (197, 29): 172.4; (400, 500): 383.9.

Node (400, 500)

- Number of visible nodes = 20: (106, 55): 533.4; (107, 54): 533.6; (113, 37): 544.7; (113, 18): 561; (237, 521): 164.3; (237, 519): 164.1; (234, 328): 239; (233, 320): 245.5; (218, 286): 280.9; (163, 198): 383.9; (631, 521): 232; (528, 507): 128.2; (524, 505): 124.1; (506, 495): 106.1; (503, 492): 103.3; (499, 487): 99.8; (492, 478): 94.6; (469, 448): 86.4; (271, 168): 356. 2; (206, 75): 467.2.

Part IV: Conclusion

Chapter 10

Conclusion

This last chapter summarises the work described and provides the context necessary to evaluate it through the description of a selection of related works. The related work is described in the first and second sections, which characterise the approaches based on two different criteria:

- The standard, whose criterion for discerning between different maps is the kind of representation that has been chosen for the map, and
- Our alternative classification. Where we propose that the initial settings of the mapping problem should be used for providing categories to the mapping approaches.

In the introductory chapter, we claimed that, traditionally, the initial settings of the map generation problem have not been considered enough. In this manner, results of different approaches can be arbitrarily dissimilar depending on, not only on the applied methods and the map representation, but mostly on the differences in the starting points they consider. As a consequence, the evaluation (or comparison) of the obtained results is not being favoured.

The aim of this chapter is to provide a framework to the approaches presented in the pervious parts as well as to analyse their contributions. In order to provide the map generation framework, we present a set of the mapping approaches that the scientific community has proposed. This selection may be far from being complete but has been chosen on the basis of their significance with respect to our own approaches. The first section in this chapter describes each of them briefly, listed under the standard classification. The second section proposes an alternative categorisation of the same set of approaches. From these references, those that are particularly related to the approaches of this thesis, are used in the third section to compare, evaluate, and comment the characteristics and contributions of our proposed approaches. Finally, the last section describes some current and future work.

10.1 The Mapping Framework: The Classical Classification

Considering the representation criterion to classify the research done about the map generation problem, there are two fundamental paradigms that are widely recognised:

- *Area-based* paradigm. Area-based maps (Lee 96) (also known as grid-based maps, metric maps, occupancy grids, or certainty grids) divide the space into distinct regions —or cells— whose size is usually constant. This implies a homogeneous space resolution with associated properties. The algorithm that adds sensor information into the grid map assigns, for each cell, some properties describing the characteristics of the environment region represented by the corresponding cell. Hence, robot's sensors are required to provide some kind of metric information. Usually, this information describes occupancy with an associated uncertainty. The way by which this uncertainty is represented in the grid yields to different classes of area-based maps:
    - Probabilistic occupancy grids. Where Probability values are combined using the Bayes rule.
    - Evidence occupancy grids. Where Belief and Plausibility values are combined using the Dempster-Shafer rule.
    - Fuzzy occupancy grids. Where Fuzzy Sets are combined applying t-norm and t-conorm operations.
- *Feature-based* paradigm. It pays more attention in obstacles rather than in free-space. Feature-based maps usually represent an environment as a list of primitive features and their properties. Very often, feature-based maps also include specifications of the relations among features, so that features can be represented as nodes in a graph and the relations correspond to the connecting arcs. These graph-based maps are usually known as Topological maps and there is a wide range of applications that use some sort of topological map representation. In general they do not require precise metric information and are more associated to biologically inspired models or computational theories of cognitive maps. Furthermore, they can

include some combination of different representation levels or models.

Although these two paradigms are widely recognised, there have been some other mapping approaches (such as annotated maps or potential fields) that cannot be naturally included in any of the previous ones. They will be briefly presented at the end of this section.

10.1.1 Area-based Paradigm

Probabilistic Occupancy Grids

Occupancy grids have originally been proposed by Moravec and Elfes (Moravec 85). They associate, to each cell in the grid, a probability distribution over the set {Occupied, Empty}. Considering, for example, sonar information, the points of the sonar beam are projected on a horizontal plane and are used to generate map information by computing a probability density function. Points inside the sonar beam imply probability of empty space and points on the beam front imply occupancy. In the map, probability values of empty cells are represented with negative values, while probabilities of occupancy have positive values. Therefore the cells in the resulting grid map take values in the interval (-1,1).

When combining information from several sonar readings, the operations performed on the empty and occupied probabilities are not symmetrical. Empty regions are simply added using a probabilistic addition formula:

$$P''_{Emp} = P_{Emp} + P'_{Emp} - (P_{Emp} \cdot P'_{Emp})$$

The occupied probabilities for a single reading are reduced in the areas where the other data suggest is empty:

$$P_{Occ} = P_{Occ} \cdot (1 - P_{Emp})$$

Then, they are normalised (to make their sum the unity):

$$P_{Occ} = \frac{P_{Occ}}{\sum P_{Occ}}$$

After this narrowing process, the occupied probabilities from each reading are combined using the addition formula:

$$P''_{Occ} = P_{Occ} + P'_{Occ} - (P_{Occ} \cdot P'_{Occ})$$

And the final occupation value attributed to a cell is given by a thresholding method:

$$P_{Cell} = \begin{cases} P_{Occ} & if\ P_{Occ} \geq P_{Emp} \\ -P_{Emp} & Otherwise \end{cases}$$

The Probabilistic sensor modelling has traditionally dominated the generation of occupancy maps (the works at (Cai 96) and (Guzzoni 97) are just two additional examples). Nevertheless, the probabilistic approach raises a number of issues which cast doubt on its value. Several authors (Lee 96, Pagac 96, Konolige 97, and Thrun 98) have commented them.

As Konolige points out, initial experiments with the occupancy grid method ignored geometric uncertainty, assuming that all sensor returns where simple reflections and ignoring the problem of beam width. Later, Elfes (Elfes 92) reformulated the method as a probabilistic Bayesian updating problem using gaussian noise with a very large variance to account for the gross errors entailed by multiple reflections. This is not realistic, since they typically give highly-correlated readings from nearby positions and moreover, gaussian distribution implies an averaging model. He also addressed the problem of geometric uncertainty associated with sensor beam width by considering target detection under all possible configurations of the environment, but this grows exponentially with the covered area.

The work by Konolige claims that, in order to improve the probabilistic grid maps, it is necessary to define sensor models including specularity (see Sect. 1.1.4). In his work, Konolige tries to solve this problem by dividing sonar range readings between those that are from specular reflection and those from diffuse reflection. The division is done considering the probability of specularity of each individual range reading, which is computed using local information.

When defining grid maps, Elfes noted that occupancy grids are Markov random fields of order zero (that is, independent) and stated that it would be possible to use 'computationally more expensive estimation procedures' for higher-order Markov fields. Nevertheless, no examples of such higher order probabilistic grid maps seem to have been published. This problematic independence assumption has been repeatedly pointed out by Lee, Konolige, and Thrun because it can lead to significant errors. On the one hand, independence cannot be assumed when considering redundant readings: traditionally, the measured probability of a particular cell being occupied is increased by repeating exactly the same sensor reading from the same location. This is unrealistic since, in practice, the sensor reading is almost totally determined by the physical environment around the robot. Therefore, those two readings should give almost the same result as just one. In his work, Konolige tries to correct this problem by treating sensor readings independent only if they come from different robot poses (i.e., positions and orientations). On the other hand, sensor readings cannot be assumed to be independent because a sensor reading gives information about the combined probability of occupancy of a set of cells, not just a single cell. Furthermore, if a specified cell is known to be empty, this increases the probability that its neighbours are also empty. (Likewise, if the first and third cells in a line are known to be occupied, then it is more likely that the intermediate second cell is also occupied). Thrun treats this problem by means of neural networks that map sonar measures into occupancy values interpreting sensor readings in the context of their neighbours.

Since robots start with no knowledge of the objects in the environment, choosing all the initial and conditional probabilities required for the Bayesian method cause difficulties in building a reliable map (Lee 96). Moreover, since the Bayesian theory requires P_i(occupied)+P_i(empty)=1, each cell in the map is thus initialised to P_i(occupied)=P_i(empty)=0.5. And this does not express ignorance. Contrarily, it is the same as saying: "with 50% certainty, the cell *i* is occupied" before collecting any sensor readings. Finally, as Pagac points out, there is a general inability to quantify the amount and the quality of the information contained in the map when attempting to reduce the map to essentially a binary map as required for navigation. For example, cells having probabilities close to uncertainty such as P_i(occupied)=0.55, P_i(empty)=0.45 are not suitable for solving planning problems.

Evidential Occupancy Grids

Pagac, Nebot, and Durrant-White (Pagac 96) were the first in applying the Dempster-Shafer theory of Evidence for map building using occupancy grids. Their work diverges from the Bayesian approach because they allow to support more than one proposition at a time and treat probabilities as evidence. From the sonar sensors, they assign an occupancy probability distribution over the sensor arc, and since no evidence exists about the negation of this occupancy probability, the emptiness probability remains equal to 0. Similarly, the only distribution generated over the apparently unoccupied sonar sector is the one corresponding to the emptiness probability. The main advantage of this model is that it does not longer require full description of conditional (or prior) probabilities.

Yamauchi (Yamauchi 98) is another example of the generation of evidential occupancy grids. In order to reduce specular reflections, he has developed a technique called laser-limited sonar: if the laser returns a range reading less than the sonar reading, he updates the evidence grid as if the sonar had returned the range indicated by the laser (in addition to marking the cells actually returned by the laser as occupied).

Fuzzy Occupancy Grids

Fuzzy theory (see Sect. 1.1) is a particular case of the Evidential theory. Hence, the reasons that justify their use to represent occupancy information are basically the same. These reasons not only appear thorough this thesis but they have been also noted in several publications (as, for example Kim 94, Saffiotti 97, or Fabrizi 99):

- First, the stochastic method behind probabilistic occupancy grids relies on the assumption that a large number of well distributed data is available, which is rarely the case during robot navigation. On the contrary, Fuzzy logic neither requires prior nor conditional probabilities.
- Second, the fuzzy approach only needs a qualitative model of the sonar sensors, as opposed to the stochastic needed for Probability based techniques. Fuzzy logic provides an efficient tool for managing the uncertainty introduced by the sensing process. Probability theory originates in a frequentistic interpretation, while Possibility theory (which axiomatically departs from Probability theory) corresponds more to the evaluation of the ease of attainment or of the feasibility of events (Zadeh 92).
- Finally, a Possibility distribution can distinguish between the state of a cell *c* being uncertain (π_c(occupied) = π_c(free) = 0.5) from the state of being unknown (π_c(occupied) = π_c(free) = 1) —that is, when *c* has not being explored.

In their work, Fabrizi, Oriolo, and Ullivi (Fabrizi_99) define the empty and occupied space as two fuzzy sets over the universal set —the environment— assumed to be a grid. The corresponding membership functions quantify the degree of belief that each cell inside the scanning area is empty or occupied, as computed on the basis of the available measures. They combine information from sonar and a structured light vision system (see Sect. 1.1.4) using a 'winner takes all' mechanism, based on the analysis of the reasons for possible discordances and complementary. The resulting distributions convey independent information.

The second part of this thesis presents a research on Possibilistic occupancy grids that belongs to this category of fuzzy occupancy grids. In our approach (López-Sánchez 97a), free space and occupancy information comes from several robots, and a host computer generates a grid map of the environment. In the grid, each cell contains a Necessity and Possibility occupancy degree. And values are combined using standard fuzzy t-norm (*min*) and t-conorm (*max* and *probabilistic sum*) operations.

Elevation Maps

When we have introduced the area-based paradigm, we have commented that most of them contain occupancy information. However, there are some terrain mapping applications that do not distinguish between occupied and empty space but different terrain elevations. These applications have been mostly applied for the map generation of highly irregular surfaces as seafloor or planetary terrain. The main difference of elevation maps with respect to traditional 2-dimensional occupancy grid maps is that the resulting representation is 3-dimensional.

An example of research on elevation maps for seafloor was developed by Johnson and Hebert (Johnson 96). They extract the elevation map from side-scan sonar backscatter images and propose an algorithm that decreases the average elevation error. Initially, this algorithm uses sparse bathymetric data to generate an estimate for the elevation map. The initial map is then iteratively refined to fit the backscatter image by minimising a global error functional.

On the other hand, Krotkov and Hoffman (Krotkov 94) proposed a richer representation for the elevation map of a planetary terrain. Their approach assigns three different labels to each elevation point: unknown, occluded by another object, or known. When a known label is assigned to a point, it is also specified both its elevation and the robot state (i.e., position and legs configuration). In their case, since sonar sensors are not suitable for planetary use, they utilise a laser range finder to obtain the information.

10.1.2 Feature-based Paradigm

Inside the feature-based paradigm, maps have been usually presented as representations that do not require metric information because they tend to focus on features or objects rather than on the space they occupy. Nevertheless, it is rather common that they contain some kind of implicit metrics in order to enrich, support, or guide qualitative techniques. Here, we distinguish different feature-based map approaches depending on how do they organise and represent the information.

10.1.2.1 Maps of Geometric Features

In general, geometric-feature maps contain a set of features that have been specified containing some information about its shape and location with associated uncertainty.

In 1994, Cox and Leonard (Cox 94) proposed a rather theoretical approximation that consists on constructing probabilistic trees to represent different alternative models of the environment. The branches of such a hypothesis tree represent different possible measurements of assignments to geometric features. Each geometric feature has an associated covariance that models the uncertainty due to noise of the ultrasonic range data. Bayesian data association allows to calculate the probability of each hypothesis in order to prune the hypothesis tree.

The work presented in (Kim 94) describes geometric primitives of the environment as parameter vectors (in parameter space). The main difference is the representation of the associated uncertainty, which is specified by assigning appropriate fuzzy numbers to the parameter vectors. In addition, the uncertainty in the transformations (translations and rotations) between co-ordinate frames are treated through fuzzy arithmetic.

Geometric-feature maps representing the environment based on simple shapes as segments or circles are also very common. Some probabilistic examples can be found at (Vandorpe 96, Betgé-Brezetz 96 or Thrun 97). The work in (Vandorpe 96) proposes a dynamic map building representing maps as segments and circles: matching segments are updated using their associated variances in a static Kalman filter, and circles are considered to match if the distance between their centre points are smaller than a threshold. Thrun (Thrun 97) uses segments as well, although in this case, they are considered to be orthogonal. And finally, the approach described in (Betgé-Brezetz 96) represents objects (such as rocks in spatial terrain) by a location with an associated variance-covariance matrix and a shape approximated by ellipsoids. This model is built based on laser range finder data and updated using an extended Kalman filter. In addition, they generate landmarks from rocks with salient points.

Fuzzy logic has also been used in the representation of the uncertainty associated with segments. The first part of this thesis describes our approach based on imprecise segments, which contain metric information and associated fuzzy sets. The map is incrementally generated by fusing imprecise segment information that comes from different robots (López-Sánchez_98d). In addition, in our application, a host computer reasons about segment relations in order to define higher level concepts such as doors or corners. This fuzzy segment representation has been also used by Gasós (Gasós 99), who groups consecutive sensor readings into fuzzy segments in order to obtain single boundaries in the map representation. He assumes a vague previous knowledge on the objects' sizes and locations that is expressed by linguistic terms (which are provided by humans).

10.1.2.2 Topological Maps

Topological maps include a wide variety of mapping approaches representing features and their relations. In general, they tend to be closer to the biological and psychological approaches of cognitive representations.

Kuipers pioneered the work in the topological approach field. Since 1974, he tried to model the cognitive mapping capabilities exhibited by humans. In 1988, Kuipers and Levitt (Kuipers 88) established the bases for qualitative navigation, multi-level representation, and biologically inspired models.

They developed the Qualitative Navigation model: a multi-level representation theory of large-scale space based on the observation and reacquisition of distinctive visual events, that is, landmarks. This model includes a topological representation together with available metric knowledge of relative or absolute angles and distances. The topological map consists of a network of fixed features of the environment (places, paths, and regions) and topological relations among them (such as connectivity, order, and containment) defined in terms of sensorimotor experience. The quantitative information acquired during travel is assimilated into two levels of metric description: local geometry and orientation frames.

Kuipers and Levitt also defined the concept of 'view' as the sensory image received by the observer at a particular point. In this manner, views could be used to recognise places, and paths could be represented by means of sequences of actions and views. This concept of view is very relevant in biology because there are neurones in the brain (place-cells in the hippocampus) that fire only within the context of a particular view. However, they treated views as opaque objects that can be used as indexes for associative retrieval of landmarks in a memory. Views are not necessarily visual, but must be distinctive enough to allow the assimilation of the environmental structure. In this approach algorithms were developed under the assumption of correct association of landmarks on reacquisition.

Maps Based on Graphs

Every topological map requires some kind of structure to represent features of the environment and their relations. Probably, the one that comes more naturally is the graph representation. Here, we present some of the approaches to topological maps that use some form of graph as a basis for their structure.

Mataric's approach (Mataric 90) defines topological maps that are graphs having landmarks as nodes and arcs connecting neighbour landmarks. Straight wall segments are used as landmarks whose recognition involves robot motion. In addition, this approach includes dis tance information to help in the localisation of the robot.

Lu and Milios [Lu_97] propose a completely different approach. They use the graph representation to study the problem of consistent registration of multiple frames of measurements. In the nodes, they include local frames of range scan data together with the corresponding estimated poses of the robot (that is, robot's position and orientation). The relative spatial relationships between local frames are represented by two kinds of links. Weak links relate adjacent poses along robot's paths. And strong links join local frames whose scan data have 'sufficient' overlap (the spatial extent in the overlapping part of two scans should be larger than a fixed percentage of the spatial extent covered by both scans). Consistency is achieved by using all the spatial relations as constraints in order to solve all the data frame poses simultaneously.

There are also specific kinds of graphs that can be used for representing an environment. For example, view graphs (Franz 97) relating scene views in outdoors environments, or visibility graphs (see Sect. 8.2.1) in polygonal terrain. The latter, has been used in the rather theoretical approach described in (Rao 96), which studies the performance of several point-robots in order to reduce the sensing time. The proposed robots are point-sized. They are equipped with vision sensors that return the visibility polygon associated to their position (that is, all points that can be seen from that position).

In topological approximations, the poor metric sensing must be compensated by better pattern matching. This means that they are strongly dependent on the recognition of previously known landmarks. For example, the approach by Shatkay and Kaelbling (Shatkay 97) gives a method that learns topological map from landmark observations. It considers local topological information along with some landmark information to disambiguate different locations. The method is based on a recursive estimation routine that can refine positions estimates backwards in time. This approach extends a previous one by Koeing and Simmons (Koeing 96), who investigated the problem of learning the distances in a topological map if a topological sketch of the environment was readily available. Their approach is to provide the robot with the topological and geometrical constraints that are easily obtainable by humans, and have the robot to learn the rest. They use probability distributions to create a model for optimal decision making (Partially Observable Markov Decision Process or POMDP) that incorporates the distance uncertainty as well as sensor and actuator models. The POMDP is specified as a set of states, a set of actions for each state that can be executed with a transition probability, and the probability that each sensor reports the correct features of the current state. Each state specifies robot's location —with 1 m of resolution— and orientation (considering only 4 compass directions).

Multi-Level Maps

Whenever one has to deal with a significant amount of data, there is a tendency to group information and to represent it with a higher level of abstraction. Here, we present some examples of topological approaches that organise their information in multi-level map representations.

Levitt and Lawton (Levitt 90) developed a two-level theory of spatial representation of the environment based on the recognition of landmarks. In the first level, viewframes constitute regions of the space defined by relative angles and estimated distances. In the second level, every two landmarks create the so called Landmark Pair Boundaries, lines that lead to topological divisions of ground space into regions (orientation regions). The robot navigates by moving between orientation regions. Navigation can be done qualitatively by crossing the boundary lines.

The multi-level approach representation of Kortenkamp (Kortenkamp 93) is based on a model of the human cognitive mapping. Kortenkamp adapted it for implementation on a mobile robot. The model consists of 3 components that lead to two different representation levels. The first component is a Bayesian network that performs place recognition from sonar and vision information. The second component uses spreading activation networks that encode direction and familiarity. This component represents regional networks (topological maps) and allow to select a route to a goal as well as directions to the next place (gateways) along that route. Finally, the third component represents the regional map (survey map for the psychologists) that creates a global image-like overview of its environment from the abstraction of its local representations.

Environmental information can also be grouped hierarchically. For example, Bulata and Devy (Bulata 96) propose a hierarchical model with three different levels. The first level corresponds to the geometrical model that describes landmarks. Landmarks are extracted from sensory data using a segmentation algorithm upon the acquired points. Extracted landmarks are then compared to the models by means of rules. Landmark models correspond to characteristic local feature groupings as corners or doors. From the extracted landmarks, only are selected those that are accurate enough to be recognised and located. The second level is the semantic level and is defined by grouping landmarks that can be perceived from the same robot position. These correlated landmarks form area models and are the base for the third level of this approach: the topological level. This is an environment model and describes semantical areas such as room or corridor with perceptual constraints. This model is built from the relationships between area frames, it is represented by a random vector and a covariance matrix, updated through the use of an Extended Kalman Filter.

Simpler approaches have been also proposed. For example, Hébert, Betgé-Brezetz, and Chatila (Hébert 96) used odometry to relate local information in a global representation. Local information is extracted from sensing data, so that local maps contain perception-related objects. On the other hand, odometry is decoupled from exteroceptive information in order to provide a global frame relating local maps in the global world map.

The third approach that this thesis presents could be considered to be a hybrid approach between map representations based on graphs and this multi-level approach (López-Sánchez 98a). In this manner, we group polygonal information into obstacle areas and associate a visibility graph with the level of information we are considering in the path planning task. Nevertheless this approach diverges from most topological map representations because it does not involve any landmark recognition process.

Biologically Inspired Approaches

Biologically inspired approaches mostly refer to neural-network-like applications. Despite their scientific interest (they try to help in the understanding of the mapping process in biological creatures), they are less related to the approaches presented in this thesis, and therefore, we just briefly comment a few significant approaches (they are by no means complete).

Zimmer (Zimmer 96) uses neural networks to relate situations (that is, records of sensor information together with their approximate position) by means of an adjacency relation criterion.

Prescott (Prescott 96) presents a network architecture in which continuously changing activations of the units can be viewed as a dynamic map that is permanently oriented to the position and heading of the agent. This approach maintains multiple, partial, and overlapping models of the environment based on the barycentric co-ordinate frames defined by groups of salient landmarks.

Zipser (Zipser 86) develops a 2-layer neural net. The first layer corresponds to the sensory system. It detects the landmarks and stores descriptions that can be recognised afterwards. The second layer is in charge of giving the similarity between the current robot scene and the scene as recorded at the centre of a place-field. Place-fields are locations relative to a set of distal landmarks. In his approach, Zipser also presents two additional models to locate goals: a three-layer neural net, and the so called β model. Nevertheless, these two models are less based on neurophysiological experiments. Prescott has used the latter model (i.e., β-model) in his previously described work. The β-model corresponds to a two-layer network that records information about three landmarks and the goal in the so called β weights. This information is recorded once, and can be used to locate the goal location unambiguously anywhere the landmarks are visible.

10.1.3 Grid Maps versus Topological Maps

Many authors have compared these two complementary approaches to robot mapping. Such comparisons can be found in (Lee 96, Prescott 96, Saffiotti 97, Konolige 97, Thrun 98a, or López-Sánchez 98a) among others.

In general, grid maps are considered to be easy to construct and maintain and their representation naturally reflects neighbourhood relations. Grid maps rely basically on odometric information and their main problem is that their representation require a significant amount of memory. On the other hand, topological maps have a much more compact representation. They require a more elaborated update but provide abstract representations of the environment that favour reasoning and information treatment processes. Topological maps are biologically inspired approaches that are based on landmarks for both map generation and robot positioning tasks. Nevertheless, positioning is very sensitive to the recognition, number, density and uniformity of distribution of landmarks.

A widely treated problem in map generation is to establish correspondence between current and past locations. This allows to fuse information about the same feature, to follow paths in a previously explored environment or to localise the robot. In grid-based mapping approaches, the correspondence problem is attacked exclusively through metric information: if the robot is capable of accurately estimating its co-ordinates in a Cartesian co-ordinate frame, the correspondence problem is solved. However, in most cases the positioning system accumulates error, and thus, some inaccuracy model must be included. Additionally, external sensor information may be also used to refine the position estimate by matching the currently perceived local grid with the global map. In fact, topological approaches typically use this last method. They compare external information coming from the robot's sensors with some expected characteristics: landmarks or special configurations of landmarks.

Grid maps can be considered to represent low-level features, whilst topological maps concentrate more on representing high-level features. The suitability of both approaches is highly dependent on the environment and the robot sensors. High level features may make the map more robust, as these features are more stable over time. However, high-level features might appear in a low density distribution and might be difficult to extract and recognise.

Topological approaches are based on computational theories that model human cognitive mapping. The biggest difficulty in applying the ideas from these computational models to robots is that they heavily rely on perception, particularly perception of landmarks. The Kalman filter has been widely used to treat uncertainty in the landmark recognition process. It gives the optimal update for the robot and target positions having noise in sensor readings and robot positioning. The problem is that choosing the wrong target can lead to divergence of the Kalman filter so that the robot becomes lost in the environment. Cox and Leonard (Cox 94) point out the importance of this problem, and suggest a Bayesian tree approach to formulating and processing multiple hypothesis about data associations.

Topological approaches try to avoid the use of odometric information for the disambiguation of landmarks by using a short history of sensor inputs or specific navigation routines. Nevertheless, it sometimes becomes necessary and many of them use some kind of implicit metric information. This is specially the case for robots equipped with sonar sensors, for whom most of the environment looks alike.

Hybrid Approaches

Taking into account that grid and topological maps are complementary approaches that exhibit orthogonal advantages and disadvantages, several authors have proposed methods for combining both paradigms with the aim of compensating the weaknesses of one method with the strengths of the other.

In the second approach of this thesis we extract a graph representing free-space from our possibilistic grid representation (see Sect. 8.2.1). This graph is a visibility graph, where nodes correspond to obstacle vertexes and arcs join visible (non-obstructed) nodes. It allows to plan optimal paths and specify them as abbreviated sequences of intermediate points. Other approaches, as for example the one by Yamauchi (Yamauchi 98), create graphs from the grid representation transforming cells representing free-space directly into nodes of the graph. Cell adjacency is then translated into arcs joining the corresponding nodes. The algorithms applied to plan paths can be equally applied, the main difference is the number of nodes that must be treated and used to specify the resulting paths.

Another approach that combines both paradigms has been proposed by Thrun (Thrun 98a). It uses neural networks to learn the mapping from sensors to occupancy values. Neural networks interpret sensor readings in the context of their neighbours, thus increasing the accuracy of the occupancy information. Afterwards, a Bayesian integration over time yields the resulting grid map. And a topological map specifying regions in the environment is then generated on top of the grid-based map. This is accomplished by using thresholding methods and obtaining a Voronoi diagram (see Sect. 8.2.1).

The opposite idea is applied by Lee (Lee 96). He proposes to derive the grid-based map from a feature-based map. Using information coming from ultrasonic sensors, the former map is created by detecting and clustering potential features. These features are updated in the feature-based map, whose information is afterwards included into a probabilistic grid representation. He proposes this order on the bases of the problems derived from independence assumptions and erroneous interpretations that appear when using probabilities in a early stage. (We have already discussed these problems when discussing the probabilistic occupancy grids).

Finally, just comment that some algorithms for the conversion between the two mapping paradigms have been proposed by Horst (Horst 96). His algorithms convert spatial information in certainty grids into object boundary curves, and vice versa.

10.1.4 Other Map Generation Approaches

Although the classification of area-based maps and feature-based maps can include most of the map generation approaches for robots, it is worth noting that there are some map representations that do not naturally fit in this classification. Here, we just briefly comment some of them.

An annotated map (Thorpe 90) is a 2-dimensional representation that includes spatial information as well as procedural information. This procedural information triggers certain events in the robot. Annotated maps are effective because they tie procedural information to spatial locations, thus making the procedural information easy to organise and retrieve (Kortenkamp 93).

Homing (Nelson 89) constitutes the less complex mapping strategy. In homing, the robot does not explicitly construct the map, it stores sensory data about the environment and associates movements with sensory events. As sensory events trigger movements, the robot navigates the environment.

Rekleitis, Dudek, and Milios (Rekleitis 97) propose a systematic method to cover free space with trapezoids using two robots that collaborate in order to reduce odometry errors. The two robots are equipped with object detector sensors and a robot tracker, which locates the other robot accurately. Robots move one at a time: the moving robot explores, and the stationary one reports the distance and orientation of the moving robot. Exploration consists of two logical parts: local exploration (which sweeps a horizontal stripe of free space inside one trapezoid), and global exploration (which connects the straps together and decides which part to explore next). The order in which stripes are explored is given by a depth first search algorithm.

A similar exploration based on a surface filling algorithm is presented in (González 96). It represents the environment using adjacent regions that have been already explored and the robot (that uses sonar sensors) moves towards the boundaries with the unknown area.

The map construction in (Sgouros 96) accepts as input a topological diagram of the environment along with the placement of the proximity sensors. The environment is discretized and at each point it is computed which sensors are active. In this manner, the map describes qualitative variations in sensor behaviour between adjacent regions. This map also includes attractors that are defined for doorways and goal points, so that it is used to specify preferred motion directions based on the topological relation between each point.

Finally, the approach by Liu and Wu (Liu 99) presents an evolutionary group of robots that builds a potential field map (defined in (Hwang 92)) of an unknown environment. The robots directly interact with the physical environment and a remote host handles the computation of the global map based on the information received from the robots. From the proximity measurement of each robot at a certain location, it is estimated the proximity of neighbouring locations to sensed obstacles. They express the confidence on the proximity estimates by weighting each measurement with a function of the distance between robot's position and the neighbouring locations.

10.2 Our Proposed Characterisation

Up to this point we have commented more than forty approaches to the map generation problem. As we have seen, most of them can be classified using a criterion that discerns between approaches depending on the map representation they use. In this thesis, in addition to providing three new approaches, we claim that all the initial settings of the mapping problem should be taken into more consideration, both when defining the problem and also when comparing the approaches. In the Introduction Chapter, we provided a list of the problem settings that we consider are essential to define a mapping approach (see Sect. 1.2.2.1). We think that the results of an approach are subjected to the initial settings of the problem and only those approaches assuming similar conditions can be compared. Hence, we propose not only to use the map representation criterion when classifying the mapping approaches but to consider all the initial settings to characterise the map generation approaches. In this manner, for example, the criterion of the kind of environment can be used to discriminate among approaches that are more suitable for indoors, outdoors, planetary, or underwater terrain.

This section uses the settings listed in Sect. 1.2.2.1 to define the classification that each of them implies. For each classification we will reference those approaches (from the ones cited in the previous section) that fit more naturally the given classification. This will also help to find the approaches related to the ones proposed in this thesis, which will be discussed in the next section.

Kind of Terrain

We have already commented that the kind of terrain that must be modelled implies the following classification for the mapping approaches:

- *Indoors*: Indoor environments are usually considered to be flat and structured, that is, containing perpendicular walls defining corners, parallel walls defining corridors, straight features as sides of pieces of furniture, etc. In general, their dimensions are in terms of meters and for this reason most occupancy grid applications consider indoor environments. Nevertheless a significant number of topological approximations also consider indoor environments. Some examples of indoor maps can be found in (Moravec 85, Kortenkamp 93, López de Màntaras 97, Thrun 98a, or López-Sánchez 98c).
- *Outdoors*. Outdoor environments have different characteristics than indoors. In general, they present an irregular terrain, although it is supposed to be smooth enough to allow robot's displacement. The features are distributed with a rather low density and can be of any kind: buildings, trees, mountains, roads, etc. Usually, most of them are pre-acquired in order to help in their recognition. Therefore, mappings based on landmarks (as the ones proposed in (Kuipers 88) and (Levitt 90)) are specially suitable for this kind of environments.
- *Planetary*. Although planetary terrain may look similar to that in outdoors, it has several characteristics that require a special treatment. On the one hand, the terrain is usually rugged and irregular (with a relatively large density of sharp rocks), and hence, the geometry of the terrain must be accurately modelled. And, on the other hand, the fact that it belongs to the space means that laser range finders are the most appropriated sensors (as the one used in (Krotkov 94)), although some approaches also use cameras (Betgé-Brezetz 96).
- *Underwater*. Finally, the underwater environment constitutes a very particular environment where the robot, its sensors, and movements must be specially adapted. In this thesis we have only commented the seafloor map generation in (Johnson 96), which is focused in obtaining an elevation map.

Previous knowledge of the Environment

The previous information that the robot has about the environment is a key aspect of the mapping problem. As we have already commented in the Introduction Chapter, it implies a redefinition of the map generation problem.

- *Partially known*. When the robot has been provided with an approximated sketch of the environment, the map generation process consists in refining this information while navigating. The approach in (Koeing 96) is to provide the robot with the topological and geometrical constraints that are easily obtainable by humans (i.e., a topological sketch). From this information, the robot has to learn the distances while learning to navigate in the environment. This approach provides a probabilistic model for optimal decision making that incorporates the distance uncertainty and the sensor and actuator models. Similarly, the approach presented in (Sgouros 96) accepts as input a topological diagram of the environment. In 1996, one of the events of the AAAI Mobile Robot Competition and Exhibition (explained in (Kortenkamp 97)) consisted in an office navigation, for which the robots were given a graphic representation of the office building, showing rooms and hallways and rough distances. This year, a multi-robot approach explained in (Guzzoni 97) won.
- *Previous information about some objects*. When the robot has been given information about landmarks, to solve the map generation problem means to recognise these landmarks and to establish relations between them. By landmarks we understand both artificial beacons added to the environment or objects existing in the environment about which the robot has some information (a model, an image, an approximate description, etc.) that helps in its recognition. For example, in (Levitt 90) a tree, a mountain, and a building are supposed to be recognisable. More moderated assumptions are supposed in (Bulata 96), an approach that extracts landmarks (such as corners or doors) from sensory data (segmentation algorithm upon the acquired points) and compare them to certain models by means of rules. Similarly, Gasós (Gasós 99) proposes to provide the robot with linguistic descriptions about the objects and their distribution in an specific environment. In this manner, the robot can match sensor information with these linguistic descriptions in order to generate a geometrical map of the environment. Finally, landmarks can also be specified as positions associated with sensor information.

In some approaches (as the ones by Thorpe or Thrun) these landmarks are indicated by humans. In the case of the (Thrun 98b), a human that drives with joy-stick the robot, chooses and marks a collection of significant locations in the robot's environment that are afterwards used as landmarks.

- *Completely unknown*. In general, the less information the robot has about the environment, the harder it is the task of generating its map. Without previous knowledge, there is not a direct way for distinguishing the relevant information and, therefore, every discovered feature is added to the map. Nevertheless, there is always a minimum amount of information about the kind of environment (indoors or outdoors) or the density of obstacles. This is because such information is required to select the map representation. Some examples of approaches that map unknown environments are (Vandorpe 96) and (López-Sánchez 98c). Finally, there are approaches that assume some additional characteristics about the environment. For example, (Thrun 97) and (López-Sánchez 97a) assume vertical and horizontal walls.

Exploration and Number of Robots

There are two main aspects of the information gathering process that define different robotic map generation approaches: Who performs the exploration? and How is the exploration performed?

The latter aspect refers to the exploration strategy. Robots can explore the environment by following a random strategy (Amat 95, López-Sánchez 97b), moving towards the boundaries with the non-explored areas (González 96, Yamauchi 98), by following given paths (Kuipers 88), or even driven by humans (Thrun 98b). Nevertheless, there are a significant number of approaches that consider a sequence of points in the environment (where the robot stops and senses) that do not specify how this points are selected. These approaches are usually more concerned with the stationary data acquisition. The approaches in (Pagac 96) or in (Oriollo 98) are just two examples.

To the question "Who performs the exploration?" we answer with the number of exploratory robots. Therefore, we distinguish between:

- *Single-robot mapping approaches,* and
- *Multi-robot mapping approaches*

Most of the approaches that we have commented consist of a single robot exploring the environment. Multi-robot systems have been used for a variety of tasks such as interaction and learning (Mataric 94), soccer competitions (Shen 98, Veloso 98), foraging (Balch 99), positioning

(Kurazume 96), adaptive co-operation (Ghanea-Hercock 99), or co-operation between heterogeneous robots (Parker 94). However, it has not been extensively used in the mapping task.

As we already commented in the Introduction chapter (see Sect. 1.2.1), exploring the environment with several robots increases the robustness of the process: the accumulation of odometry error becomes distributed and the acquired information degrades proportionally to the number of unsuccessful robots.

The multi-robot map generation issue is a rather new idea. In 1995, we proposed the idea of the first part of this thesis: to have several small autonomous robots exploring the environment and a host computer to build the map (Amat 95). The same schema has been used in (Guzzoni 97), (Liu 99) and, in general, all the approaches having simple robots. Obviously, there are also approaches that do not require a base computer, as the one presented in the third part (López-Sánchez 98a) (which, in addition, uses an heterogeneous group of robots). Some of them do not require a high level of co-operation, so that the results obtained by one robot can be extended to more than one robot, which is the case of (Thrun 98b). On the contrary, others present a tight collaboration, as in (Rekleitis 97), where one robot moves while the other keeps track of it without moving.

There are also some rather theoretical approaches to multi-robot mapping, as the ones in (Rao 96) or (Cai 96). The former reduces the sensing time for point-sized robots. The latter, integrates the sensing information of all the robots considering their corresponding reliability.

The approach by Yamauchi (Yamauchi 98) is particularly interesting because it provides an exploration strategy for a group of robots. The map is represented by a evidence grid. Free-space cells that are adjacent to unknown cells are considered as frontiers, and the robots explore by navigating towards these frontiers. The path planing is a depth-first search on the free-space cells having as goal one of the frontier cells. Whenever a robot arrives at a new frontier, it sweeps its sensors and constructs a local evidence grid representing its current surroundings. This local grid is integrated with the robot's global grid, and also broadcast to all of the other robots grids. When the robot reaches its destination, that location is added to the list of previously visited frontiers. Hence, several robots may waste time by navigating to the same frontier. If a robot is unable to make progress towards its destination for a certain amount of time, then the robot will determine that the destination is inaccessible, and its location will be added to the list of inaccessible frontiers. In addition, if it is the case that one robot blocks another, the blocked robot will mark the frontier as inaccessible, so it will be explored only if other robots explore it. And finally, detected robots can be also included as representations of occupied space, however, their locations may be afterwards rescanned.

Sensors

The robot's equipment characteristics are another crucial aspect that must be considered when defining a robot mapping approach. This is because the nature of the sensors not only provide the input to the map generation process but also determine the kind of uncertainty associated with the gathered information.

In the Introduction Chapter we already described the most commonly used sensors, their characteristics and the kind of uncertainty they generate (see Sect. 1.1.4). Moreover, in the previous section we have commented how do they have been modelled in many probabilistic, evidential or fuzzy approaches. Therefore, here we just list several of the previously referenced works based on the sensors they use:

- Sonar: (Moravec 85), (González 96), (Pagac 96), (Konolige 97).
- Camera: (Franz 97).
- Laser range-finder: (Lu 97), (Vandorpe 96), (Krotkov 94), (Betgé-Brezetz 96), (Bulata_96).
- Infra-red: (Amat 95) (López de Màntaras 97), (López-Sánchez 97ab, 98bcd, 99ab).
- Combination of sensors:
    - camera and sonar: Kortenkamp (Kortenkamp 93) integrates sonar and vision sensing using a Bayesian network to perform place recognition.
    - laser and sonar: (Fabrizi 99) (Yamauchi 98)
    - robot sensor and sonar (Rekleitis 97), light sensor and sonar (Zimmer 96), and compass and sonar (Kuipers 88). However, the information coming from these sensors is not integrated.

Fabrizi (Fabrizi 99) et al. propose a simple co-operative mechanism for validating the information provided by the laser and sonar sensors. When sensors of different nature are used, a competitive approach has been proved to be successful. A 'winner takes all' criterion is specially appropriate when the sources of uncertainty of the used sensors are complementary. In this manner, this criterion allows to choose the most reliable information under each operating conditions. Similarly, Yamauchi has developed a technique he calls 'laser-limited sonar': if the laser returns a range reading less than the sonar reading, it updates the evidence grid as if the sonar had returned the range indicated by the laser.

Map Representation and Uncertainty Treatment

Nowadays, map representation constitutes one of the most commonly used criterion for the classification of the map generation approaches. And, although we claim that it should not be the only one, its importance is unquestionable. Section 10.0 is an overview of the mapping framework where the different approaches are distributed among the area-based and the feature-based paradigms. Therefore, we do not repeat the classification that would correspond.

The first section also details how the Probabilistic, Evidential, and Fuzzy theories have been widely used to treat the uncertainty associated with the information. Here, we just list some of the approaches that use these techniques and briefly comment some other approaches that treat uncertainty by assigning reliability weights to the information.:

- Probabilistic: (Moravec 85), (Cox 94), (Konolige 97), (Thrun 98a)
- Evidential: (Pagac 96), (Yamauchi 98)
- Fuzzy: (Kim 94), (López-Sánchez 97a), (Gasós 99), (Fabrizi 99)
- Weighted: (Cai 96), (Liu 99), (López-Sánchez 98a).

In (Cai 96) the reliability of the sensed information is concerned with the robot moving speed and the property of the sensor. The approach in (Liu 99) defines a confidence weight for each measurement. The weight is a function of the distance between robot's position and the measured neighbouring location. In our third approach (López-Sánchez 98a) the weight is a function of the area of the obstacle (i.e., the area of its polygonal 2-D projection) and the reliability of the sensor.

Map Usage

Map usage is the last (but not less important) aspect that must be taken under consideration when defining mapping approaches. In the Introduction Chapter we referenced the ecological psychology theory of affordances (Gibson 79), which says that things are perceived in terms of the opportunities they afford an agent to act. In our context, this means that the robot should solely gather the information that is helpful for its ulterior task. And naturally, this information should also be organised in a representation that favours its retrieval and the usage required by the specific task.

- *Path planning.* In general, graph representations are more useful to plan paths than grids. Some topological approaches that plan paths are (Kuipers 88, Levitt 90, Kortenkamp 93). Levitt uses two path planing algorithms: A* gives a sequence of viewframe headings connecting the current location to the ultimate goal, and a qualitative path-planning algorithm implements a rule-based recursive goal-decomposition approach. Most grid-based approaches that plan paths (Guzzoni 97, Cai 96 or Yamauchi 98) transform free-space cells into nodes of a graph, and then apply any search technique: Yamauchi uses depth-first search, and Cai et al. a path planning algorithm called D*, similar to A* but with cost parameters between cells that can change during the problem solving process. Nevertheless, it is preferable to generate a more compact graph, as in the second approach of this thesis (see Chapter 8 or (López-Sánchez 99b)). There are also less common approaches, as the one in (Sgouros 96) where resulting paths are not edges in a graph but sequences of attractors.

- *Positioning*. The positioning problem appears when the robot has not accurate odometric information. As a consequence, it does not know its location with precision and has to rely on exteroceptive perception to make an hypothesis. Although it is possible localise a robot by matching the current occupancy information with an area of an occupancy grid (Yamauchi 96, Schultz 98), it has been traditionally solved using landmarks (Fukuda 96, Boley 96, Thrun 98b). Other approaches, as the one in (Saffiotti 96), use fuzzy locations.

10.3 The Mapping Approaches of this Thesis: Contributions and Related Work

Up to this point, we have seen how the map generation problem can be solved by different approaches using mobile robots. We have characterised them on the bases of their settings, that is, those basic aspects that define the environment treatment, the robot capabilities, the used representation and the subsequent use of the map.

The aim of this section is to briefly review the three approaches presented by this thesis (summarised in the Introduction Chapter), and simultaneously, to comment their contributions to the general framework as well as the commonalties they share with other approaches.

10.3.1 Representation of Orthogonal Indoor Environments by means of Fuzzy Techniques

The approach presented in the first part of this thesis generates the map of an indoors environment that is considered to be unknown but orthogonal and mainly passable. The map generation process is performed by a group of small autonomous robots that explore the environment and a host computer that generates the map incrementally from the information gathered by the robots. Robots explore the environment by moving randomly in free-space and following walls —or obstacle edges— when detected. Whenever two robots meet, they co-operate by sharing their information about the followed features. They also communicate this information to the host computer, so that it can incrementally generate the map. This map consists of a list of imprecise segments that are afterwards related to form doorways and different kinds of corners. The main source of uncertainty associated to the obtained information is the odometry error. This error has been statistically studied and used to define the fuzzy sets associated with the imprecise segments. These fuzzy sets are utilised to fuse different imprecise segments that may correspond to the same feature in the environment.

Contributions

The main contributions of this first part are the implementation of:

- *Multi-robot mapping*. We have been referenced as the first approach to collaborative multi-robot mapping (Thrun 98b). (Amat 95) describes the first publication proposing the multi-robot approach described in this thesis[11]. Nevertheless, our robots co-operate by sharing maps when they meet, and this was previously discussed by Ishioka et al. (Ishioka 93).
- *Imprecise segments*. We define an imprecise segment as a segment that represents a feature of the environment (a wall or an obstacle edge) with uncertainty about its location. This uncertainty is represented by means of a fuzzy set (see Sect. 3.2.2).
- *Segment fusion*. We provide an algorithm to combine imprecise segments that may correspond to the same feature in the environment (see Sect. 3.2.2).
- *Map completion*. After the map generation process, we propose rules for reasoning about segment relations in order to define higher level concepts such as doors, L-corners, and T-corners (see Sect. 1.1).

[11] In 1995, the research of this thesis was just starting and, although my name does not appear as an author, it is mentioned in the acknowledgements section.

Related Work

We have already commented that just a few multi-robot systems have been used for studying the map generation problem. From the approaches we have referenced and explained in the previous section, we can conclude that most of them cannot be compared to our system. Two of them (Guzzoni 97 and Liu 99) have the same layout of several robots and a base computer. Since the former is based on having a previously given graphic representation of the environment, we consider that our approach for mapping unknown environments cannot be compared with this approach. The latter (Liu 99), (which is simulated) generates potential maps that are difficult to evaluate because the approach is focused on robot learning aspects (uses genetic algorithms as a global optimisation method for selecting the reactive motion strategies of the robots).

Although the multi-robot approach in (Thrun 98b) does not provide results with more than one robot, the single-robot case can be extended to the multi-robot rather naturally. Nevertheless, this single robot explores the environment for the first time driven by a human (who, in addition indicates the landmarks that the robot will use).

Yamauchi's approach (Yamauchi 98) utilises two robots equipped with sonar sensors and a laser range-finder. Most likely, it is the multi-robot mapping approach that is closer to our research. The main difference is the way robots gather information and the implication that this fact has in the resulting map. Yamauchi's robots gather occupancy information scanning their surroundings with large-range sensors: several sonar sensors and a laser. As we have said, they combine the sensor information with a 'laser-limited sonar' technique that reduces significantly the occupancy information errors. Nevertheless, they are not completely reduced and therefore, there is still uncertainty about the existence or not of the detected features. In our case, the exploration is done by following features at a close distance. The sequence of movements necessary for a successful following guarantees the existence of the detected obstacles. Therefore, the uncertainty is about its position but not its existence. This means that we generate imprecise maps without noise (in the sense that the information is certain but not precise) whereas the ones obtained by them are noisy (that is, the information they contain cannot be guaranteed to be certain).

Regarding the map representation issue, the imprecise segments used in our approach represent their uncertainty by means of associated fuzzy sets. This same concept of fuzzy segments has been recently utilised by Gasós (Gasós 99), who groups consecutive sensor readings into fuzzy segments in order to obtain single boundaries in the map representation. The difference is that he assumes a vague previous knowledge on the objects' sizes and locations that is expressed by means of linguistic terms (which are provided by humans).

We have also provided an operation to combine imprecise segments called fusion. It is based on the combination of the fuzzy sets associated to the imprecise segments and has a relative distance threshold as parameter. Segment fusion can be only applied for imprecise segments with overlapping fuzzy sets. A similar idea is applied in (Lu 97), which links overlapping scanned areas if they have 'sufficient' overlap (a percentage threshold over the spatial extent of the overlapping area relative to the extent of both scanned areas). Similarly, the approach by Vandorpe et al. (Vandorpe 96) also matches geometric primitives using distance thresholds (as for example, for the distance between the centre points of circles).

Imprecise segment representation allows the host to reason about their relations in order to define higher level concepts such as doors or corners. We have called this process map completion and is based on If-Then rules that consider relative distances to relate neighbouring segments. Map completion is related to the approach in (Bulata 96), which uses rules to group landmarks that follow certain models. As we already noted in the completion conclusions section (see Sect. 1.1), the authors of this approach also claim that feature-based approaches using geometrical models can easily lead to a combinatory explosion.

10.3.2 Using Possibility Grids for Structured Indoor Environments

In the second part of the thesis we presented an alternative approach to the multi-robot mapping problem that is very similar to the one in the previous part (although the orthogonality requirement has been relaxed). In this case, we have developed a simulator in order to be able to control the robots in addition to mapping the environment. This allows us to design a robot navigation strategy for exploring the environment that can also be adapted to follow paths towards less explored areas. The map is a possibilistic grid representation that allows a local combination of occupancy and free-space certainty values. Paths are planned by applying a A* algorithm over a visibility graph that is extracted from the occupancy information in the grid. The environment coverage of the map can be improved by means of a wall extension process that consist on the local propagation of occupancy information in specific directions. In general, this process yields to planned paths that are safer and require less use of reactivity than paths planned over non-extended maps.

Contributions

The second part is based on the simulation of the robots of the first part. The following list solely contains the contributions that are new.

- *A simulation of the environment*. We have developed an application that can simulate several 2-Dimensional environments with groups of robots navigating inside (see Chapter 5).
- *Robots' Exploration strategy*. The simulated robots execute a simple behaviour-based navigation strategy to explore the environment randomly (see Chapter 6).
- *Possibilistic grid map*. We provide a new grid map representation based on the assignment and combination of possibility and necessity values representing ignorance, occupancy, and free-space (*see* Chapter 7).
- *Map extension*. We have implemented a method that propagates the occupancy information in the grid (using free-space and occupancy information as constraints). The method is described in Sect. 1.1, and Sect. 1.1 justifies empirically that it yields to safer paths that require less reactivity than paths planned over non-extended maps.
- *Graph extraction*. From the occupancy values on the grid it is possible to extract contours of polygonal obstacles and thus generate a visibility graph (see Sect. 8.2.1).
- *Robots' path following strategy*. We have implemented a behaviour-based path following strategy that allows the robot to follow paths (planned by the host) and to react when encountering non-previously detected obstacles. The behaviours used in path-following are very similar to those of exploration. (See Sect. 1.1).

Related Work

The introduction chapter comments that behaviour-based approaches have been widely applied to robot navigation (see Sect. 1.1.2). In our approach, each robot implements two navigation strategies: random exploration and path following. Both strategies are based on the co-ordination among different elementary behaviours. The architecture of each strategy is a deterministic finite state automaton in which each state corresponds to an elementary behaviour. It implements a "one behaviour at a time" policy, thus avoiding the problem of combination of outcomes that appears in those approaches activating more than one simple behaviour simultaneously (Saffiotti 97). Our solution is to have behaviours that are less simple (although non-complex) in order to take the decisions that satisfy the goal for each strategy.

To the best of our knowledge, there are neither other possibilistic grid approaches to the mapping problem nor grid approaches that apply any method to extent occupancy information. Nevertheless, the approach by Fabrizi et al. (Fabrizi 99) is close to ours because they use fuzzy sets over a grid. They define the empty and occupied space as two fuzzy sets over the environment (which is also unknown and office-like). The corresponding membership functions quantify the degree of belief that each cell inside the scanning area is empty or occupied, as computed on the basis of the available measures. The main difference between these two approaches is not the map representation but the robots: they use one single robot equipped with sonars and a structured light vision system (see Sect. 1.1.4) whereas we use several small robots equipped with infra-red sensors. As we have already said, multi-robot strategies are more robust and degrade gradually. Regarding the information gathering, there are two aspects to comment. Firstly, they assume the robot stops to gather information at previously known positions with complete accuracy. And secondly, the robot scans large areas at each position. They combine the sensor information analogously to Yamauchi, therefore, the discussion here is equivalent to the one in the previous subsection: their resulting maps are noisy (i.e., uncertainty about existence or not of obstacles) whilst we generate inaccurate maps (i.e., uncertainty about the precise location of obstacles).

When combining occupancy information, there are grid approaches, as the one by Konolige (Konolige 97), that distinguish independent information when it comes from different robot poses (that is, position and orientation). In our approach, we reinforce occupancy values when they come from different wall —or obstacle edge— followings. Otherwise, our approach combines occupancy values coming from the same wall following by using a *max* operation.

Finally, just comment that visibility graphs (see Sect. 8.2.1) have been also applied to other mapping approaches (Rao 96). But, as far as we know, none of them has extracted the graph from a occupancy grid. We have already commented that most grid-based approaches that plan paths (Cai 96 or Yamauchi 98) transform free-space cells into nodes of an adjacency graph. However, this method can yield to unnecessary large graphs that can be avoided by using our proposed graph extraction method.

10.3.3 Using Symbolic Grouping for Outdoor Environments

The last mapping approach we propose has been presented in the third part of this thesis. It involves a heterogeneous group robots that collaborate in the mapping of an outdoor environment. Obstacles in the environment are modelled as 2-dimensional polygonal projections (most of them are extracted from aerial images taken by an autonomous helicopter). Autonomous ground robots represent the environment by grouping overlapping obstacle polygons into higher level structures with simple shape: the obstacle areas. The resulting maps also include visibility graphs that allow to plan paths at different levels of detail (obstacle area level or polygon level). Each ground robot maintains dynamically both, its map and the path towards its goal. Finally, the uncertainty associated to the information is represented through reliability weights in the arcs of the visibility graph.

Contributions

- *Outdoor heterogeneous multi-robot mapping*. We propose a mapping approach that involves airborne and ground autonomous vehicles. They are equipped with different sensors and have different tasks: the helicopter flies towards and area taking aerial images and ground robots plan paths dynamically and detect obstacles with sonar sensors. Our outdoor map representation integrates features extracted from different sensors and different robots.
- *Robust robot co-ordination*. Each time a robot extracts a new polygonal feature, it broadcasts the information to the rest of robots. In this manner, if the communication is permanently established, all robots generate the same map. Nevertheless, since robots are completely autonomous, a communication failure just implies that robots will plan paths considering less obstacle information.
- *Dynamic path planning at two different levels of detail*. Grouping obstacle polygons into obstacle areas provides a hierarchical map representation that simplifies its update and treatment. This reduces the complexity of visibility graphs associated to each level of information.

Related Work

Most multi-robot mapping approaches rely on robot communication. Nevertheless, some of them are more robust than others. For example, in

(Rekleitis 97) full communication is assumed and required because a moving robot obtains its current position from a second observer robot at any time. Therefore, if the communication fails, the robot can hardly integrate its gathered information without knowing its own position. Other approaches, as the one by Yamauchi (Yamauchi 98) are less dependent because, like our approach, they broadcast new scanned information but robots explore completely autonomously. However, in our approach, the helicopter provides most of the information and, in general, the communication between the air and the ground is less obstructed that the communication between ground robots. Hence, as a direct consequence of having heterogeneous robots, we can expect that the robots in our approach share information more efficiently than the homogeneous robots utilised in Yamauchi's approach. To the best of our knowledge, our approach constitutes the only heterogeneous robot colonies that generates maps of unknown outdoor environments. Due to the image polygon extraction algorithm in the helicopter, we need to assume that the outdoors terrain is similar to a parking area. Nevertheless, this assumption is less informed than the one assumed in (Guzzoni 97), where two different robots navigate in an office-like environment using a previously known graphic representation of the office building (including rooms, hallways, and rough distances).

Multi-level maps and representations based on graphs have been already discussed in the Topological Maps subsection (see Sect. 10.1.2.2). From this map representation point of view our approach can be considered as a hybrid (López-Sánchez 98a). We group polygonal information into obstacle areas and associate a visibility graph with the level of information we are considering in the path planning task. From the feature based maps subsection 10.1.2, it is obvious that these ideas of grouping information and utilising graph representations are by no means new. Nevertheless our approach uses its representation at different accuracy levels to plan paths, and diverges from standard topological map representations in the fact that it does not involve any landmark treatment.

Finally, our proposed approach represents uncertainty through reliability weights in the arcs of the visibility graph. These reliability weights are computed as a function of the area of the obstacle and the reliability of the sensor. This uncertainty treatment is similar to the approaches in (Cai 96) or (Liu 99). In (Cai 96) the reliability of the sensed information is concerned with the robot moving speed and the property of the sensor. The approach in (Liu 99) defines a confidence weight for each measurement, where the weight is a function of the distance between robot's position and the measured neighbouring location.

10.4 Future Work

This thesis presents the research done with three different settings of the map generation problem. Nevertheless, we are currently working in two research projects that also require a map of the environment. One of these settings corresponds to an autonomous robot doing transportation tasks inside a factory. Since the basic distribution in a factory does not change very often, a map of the environment can be provided. Hence, the main task is to plan paths and to be able to follow them while avoiding non-permanent obstacles that may obstruct the planned paths. Currently, the robot is under construction, so that in the medium term we will be able to incorporate to the robot our behaviour-based path following approach, which has been proved to be successful under simulation.

The other project we are currently working on corresponds to a legged robot that navigates in unknown, outdoor environments. In this case, the robot is equipped with stereovision and its image processing system is able to extract several landmarks. Hence, our mapping approach is based on landmarks and, since the odometric information in a legged robot is extremely poor, the map representation must be topological. Stereovision can also provide some metric information (such as distances to landmarks) that will be incorporated in the map representation. For this approach we will follow Prescott's (Prescott 96) idea of using the β-model (proposed in (Zipser 83)) to record information about groups of three landmarks and the goal. This information is recorded once, and can be used to locate the goal unambiguously if the three landmarks are visible. In our case, since landmarks will not be assumed to be always recognisable, we will probably use some additional landmarks.

The research presented in the second part of this thesis can be extended to apply some learning techniques. In fact, during the development of this part, we studied some reinforcement learning algorithms, specially the Q-learning, which has been successfully applied to robot learning. (Murao 97, Szepesvari 97, Yoshida_98). Nevertheless, reinforcement learning algorithms require long learning periods that cannot be afforded by most autonomous robots applications that do not consider toy problems. As a consequence, we explored the field of case base reasoning. In order to learn from examples, it is necessary to define an example as a set of attribute-value pairs. In our case, since our behaviour-based strategy is based on If-Then rules, it is possible to translate these rules in cases. The variables in the rules would correspond to the attributes. In this manner, for each behaviour, we can generate two decision trees: one containing the cases that correspond to the rules that define the next action to execute, and another for deciding the activation of another behaviour. The underlying idea is that the rules that have been developed for the basic behaviours can be used as a knowledge base previous to the learning process. By considering these cases, the robot could face similar cases with similar actions. In order to do this, it will be necessary to define a similarity function to retrieve from the knowledge base the most similar case. In addition we also require a method for adapting the retrieved case into the action necessary in the current situation. Currently, the translation of the action rules into cases for the exploration behaviours has been done. Nevertheless, our robots do not learn yet because we just have used the equality as similarity function, and we do not adapt the resulting actions. We propose to face the symbol grounding problem from this point of view. For example, the wall end symbol has been already represented as a specific sequence of sensor readings in the wall following behaviour. Therefore, we can interpret this case as a grounded concept because the robot recognises it based on its own sensing and experience. Our intuition says us that similar concepts as corners or doors could also be learned and grounded, but we have not proved it.

Bibliography

[Alami_93] R. Alami, R. Chatilla, and B. Espiau "Designing an Intelligent Control Architecture for Autonomous Robots", Proceedings of the 7th *International Conference on Advanced Robotics*, 1993, Tokyo.

[Aleliunas_79] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovasz, and C. Rakoff, "Random Walks, Universal Traversal Sequences and the Complexity of Maze problems " in 20th *Annual Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, Oct 1979. pp. 218-223.

[Amat_95] J. Amat, F. Esteva, and R. López de Màntaras, "Autonomous Navigation Troop for Co-operative Modelling of Unknown Environments" in Proceedings of the *Seventh International Conference on Advanced Robotics: ICAR '95*. Vol. 1, pp 383-389.

[Arkin_87] R. C. Arkin, "Motor schema based navigation for a mobile robot: an Approach to Programming by Behaviour" in Proceedings of the *International Conference on Robotics and Automation*, 1987, pp.264-271.

[Balch_99] T. Balch, "The Impact of Diversity on Performance in Multi-robot Foraging". In Proceedings of the international conference on *Autonomous Agents*. Seattle 1999. pp. 92-98.

[Betgé-Brezetz_96] S. Betgé-Brezetz, P. Hébert, R. Chatila, and M. Devy, "Uncertain Map Making in Natural Environments". In Proceedings of the *IEEE: International Conference on Robotics and Automation,* 1996. pp 1048-1053.

[Boley_96] D. L. Boley, E. S. Steinmetz, K. T. Sutherland, "Robot Localisation from Landmarkds using Recursive Total Last Squares". In Proceedings of the *IEEE: International Conference on Robotics and Automation,* 1996. pp 1381-1386.

[Bonasso_98] R. P. Bonasso, D. Kortenkamp, and R. Murphy, "Mobile Robots: A Providing Ground for Artificial Intelligence" in Artificial

Intelligence and Mobile Robots: Case Studies of Successful Robot Systems *The AAAI Press / The MIT Press.* 1998. pp.3-18.

[Borland_94] Borland International, "Borland Object Windows: Programmers Guide v.2.5" *Borland International, Inc.*, California. 1994.

[Brady_85] M. Brady, "Artificial Intelligence and Robotics" *Artificial Intelligence*. Vol. 26, 1985. pp 79-121.

[Broder_94] A. Z. Broder, A. R. Karlin, P. Raghavan, and E. Upfal, "Trading Space for Time in Undirected *s-t* Connectivity" SIAM J. COMPUT. Vol. 23, No. 2, pp 324-334, April 1994.

[Brooks_86] R. A. Brooks, "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation* RA-2(1), 1986, pp. 14-23.

[Brooks_91] R. A. Brooks, "Intelligence Without Reason" Proceedings of the *International Joint Conference on Artificial Intelligence*, 1991, Sydney, pp. 569-595.

[Bulata_96] H. Bulata and M. Devy, "Incremental Constr uction of a Landmark-based and Topological Model for Indoor Environments by a Mobile Robot". In Proceedings of the *IEEE: International Conference on Robotics and Automation*. 1996, pp. 1054-1060.

[Cai_96] A. Cai, T. F ukuda, F. Arai, and H. Ishihara, "Co-operati ve Path Planning and Navigation Based on Distri buted Sensing". In Proceedings of the *IEEE: International Conference on Robotics and Automation*. 1996, pp. 2079-2084.

[Chandra_89] A. K. Chandra, P. Raghavan, W. L. Ruzzo, R. Smolensky, and P. Tiwari, "The Electrical Resistance of a Graph Capt ures its Commute and Cover Times" in Proceedings of the 21st ACM STOC, 1989. pp.574-586.

[Chateauneuf_94] A. Chatea uneuf, "Combination of Compati ble Belief functions and relation of specificity". In *Advances in Dempster-Shafer Theory of Evidence*, Wiley, New York, 1994, pp 97-114.

[Cox_94] I. J. Cox, J. J. Leonard, "Modelling a Dynamic Environment using a Bayesian Multiple Hypothesis Approach ". In *Artificial Intelligence* Vol. 66, 1994. pp. 311-344.

[Cuadras_84] C. M. Cuadras, "Probability and Statistical Problems" (Spanish publication) *Promociones Publicaciones Universitarias*, 1984.

[Doménech_75] J. M. Doménech, " Statistical Methods for research in Human Sciences", *Herder Publications*, Barcelona, 1975.

[Doyle_84] P. G. Doyle, and J. L. Snell, “Random Walks and Electric Networks”, *Mathematical Association of America*, Washington D.C., 1984.

[Dubois_88] D. Dubois and H. Prade, “Possibility Theory” *Plenium Press*, New York, 1988.

[Eckel_91] B. Eckel, “Apply C++” (Spanish version), *Osborne-McGraw-Hill*, Madrid, 1991.

[Fabrizi_99] E. Fabrizi, G. Oriolo, and G. Ullivi, “Accurate Map Building via Fusion of Laser and Ultrasonic Range Measures”, to appear in Fuzzy Logic Techniques for Autonomous Vehicle Navigation, edited by D. Driankov and A. Saffiotti, *Springer-Verlag*, 1999.

[Franz_97] M. O. Franz, B. Schölkopf, P. Georg, H.A. Mallot, and H. H. Bülthoff, “Learning View Graphs for Robot Navigation” in Proceedings of the *Autonomous Agents* Conference, 1997, pp. 138-147.

[Fukuda_96] T. Fukuda, Y. Yokoyama, Y. Abe, K. Tanaka, “Navigation System based on Ceiling Landmark Recognition for Autonomous Mobile Robot”. In Proceedings of the *IEEE International Conference on Robotics and Automation*”. Minneapolis 1996. pp. 1720-1725.

[Gasós_99] J. Gasós, “Integrating Linguistic Descriptions and Sensor Observations for the Navigation of Autonomous Robots”, to appear in Fuzzy Logic Techniques for Autonomous Vehicle Navigation, edited by D. Driankov and A. Saffiotti, *Springer-Verlag*, 1999.

[Ghanea-Hercock_99] R. Ghanea-Hercock and D. P. Barnes, “Disturbed Behaviour in Co-operating Autonomous Robots”. In Proceedings of the third international conference on *Autonomous Agents*, Seattle 1999. pp.84-91.

[Gibson_79] J. J. Gibson, “The Ecological Approach to Visual Perception”, Houghton Mifflin, Boston, MA.

[Godo_93] Ll. Godo, and R. López de Màntaras, "Fuzzy Logic" in *Enclyclopedia of Computer Science And Technology*. Marcel Dekker, Inc, Pennsylvania 1993. Vol. 29, pp 211-229.

[González_96] E. González, A. Suárez, C. Moreno, F. Artig ue. “Complementary Regions: A Surface Filling Algorithm ” in Proceedings of the *IEEE :International Conferen ce on Robotics and Automation ICRA*. 1996. pp. 909-914.

[Guzzoni_97] D. Guzzoni, A. Cheyer, L. Julia, and K. Konolige, “Many Robots Make Short Work ”, In *AI Magazine*, Vol. 18, n. 1, Spring 1997, pp. 55-64.

[Hearn_88] D. Hearn, M. P. Baker, "Computer Graphics". *Prentice Hall,* 1988.

[Hébert_96] P. Hébert, S. Betgé-Brezetz, and R. Chatila. "Decoupling Odometry and Exteroceptive Perception in Building a Global World Map of a Mobile Robot: The Use of Local Maps". In Proceedings of the *IEEE: International Conference on Robotics and Automation*, 1996. pp. 757-764.

[Heller_92] M. Heller, "Advanced Windows Programming", *John Wiley and Sons, Inc.* New York, 1992.

[Horst_96] J. A. Horst, "Maintaining Multi-level Planar Maps in Intelligent Systems". In Proceedings of the *IEEE: International Conference on Robotics and Automation*, 1996. pp. 1061-1066.

[Hwang_92] Y. K. Hwang and N. Ahuja. "A Potential Field Approach to path planning". *IEEE Transactions on Robotics and Automation*. Vol. 8. n. 1. Feb. 1992, pp. 23-32.

[Ilari_90] J. Ilari and C. Torras, "2D Path Planning: A Configuration Space Heuristic Approach", in *International Journal of Robotics Research*, 1990. Vol. 9, No. 1, pp. 75-91.

[Ishioka_93] K. Ishioka, K. Hiraki, and Y. Anzai, "Co-operative map generation by heterogeneous autonomous mobile robots". In Proceedings of the Workshop on Dynamically Interacting Robots. Chamberie, France, 1993. pp. 58-67.

[Johnson_96] A. E. Johnson and M. Hebert, "Seafloor Map Generation for Autonomous Underwater Vehicle Navigation". In *Autonomous Robots*, Vol. 3, 1996, pp. 145-168.

[Kim_94] W. Joo Kim, J. Hyup Ko, and M. Jin Chung, "Uncertain robot environment modelling using fuzzy numbers". In *Fuzzy Sets and Systems*. Vol. 61, 1994, pp. 53-62.

[Kiy_95] K. I. Kiy, A. V. Klimontovich, and G. A. B uivolov, "Vision-Based System for Road Following in Real Time", in Proceedings of Seventh *International Conference on Advanced Robotics ICAR*, 1995, pp 115-124.

[Koeing_96] S. Koeing and R. G. Simmons, "Passive Distance Learning for Robot Navigation", In Proceedings of The Thirteenth *International Conference on Machine Learning ICML* 1996, Italy, pp.266-274.

[Konolige_98] K. Konolige and K. Myers, "The Saphira Architecture for Autonomous Mobile Robots" chapter 9 of Artificial Intelligence and Mobile

Robots: Case Studies of Successful Robot Systems, AAAI Press/MIT Press, Ed. by D. Kortenkamp, R.P. Bonasso, and R. Murphy. 1998, pp 212-242.

[Konolige_97] K. Konolige, "Improved Occupancy Grids for Map Building". In *Autonomous Robots*. Vol. 4, 1997, pp. 351-367.

[Kortenkamp_93] D. M. Kortenkamp, "Cognitive maps for mobile robots: a representation for mapping and navigation", PhD dissertation, CS dept. University of Michigan, 1993.

[Kortenkamp_97] D. M. Kortenkamp, I. Nourbakhsh, and D. Hinkle, "The 1996 AAAI Mobile Robot Competition and Exhibition", In *AI Magazine*, Vol. 18, n. 1, Spring 1997, pp. 25-32.

[Kortenkamp_98] D. M. Kortenkamp, R. P. Bonasso, and R. Murphy, "Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems" *The AAAI Press / The MIT Press.* 1998.

[Krotkov_94] E. Krotkov and R. Hoffman, "Terrain Mapping for a Walking Planetary Rover". In *IEEE Transactions on Robotics and Automation*. Vol. 10, n.6, December 1994, pp. 728-739.

[Kuipers_88] B. J. Kuipers and T. Levitt, "Navigation and Mapping in Large Scale-Space", In *AI Magazine*, Vol. 9, n. 2, Summer 1988, pp. 25-42.

[Kurazume_96] R. Kurazume, S. Hirose, S. Nagata, and N. Sashida, "Study on Cooperative Positioning System". In Proceedings of the *IEEE International Conference on Robotics and Automation*. 1996. pp. 1421-1426.

[Lazanas_95] A. Lazanas, J. C. Latombe, "Motion Planning with Uncertainty: a Landmark Approach", *Artificial Intelligence*, Vol. 76, 1995, pp 287-317.

[Lee_96] D. Lee, "The Map Building and Exploration strategies of a Simple Sonar Equipped Mobile Robot" Ph.D. dissertation. *Cambridge University Press*, 1996.

[Levitt_90] T. S. Levitt and D. T. Lawton, "Qualitative navigation for mobile robots" in *Artificial Intelligence,* Vol. 44, 1990, pp. 305-360.

[Liu_99] J. Liu, J. Wu, "Evolutionary Group Robots for Collective World Modelling". In Proceedings of the *Autonomous Agents*. Seattle, USA, 1999, pp. 48-55.

[López-de-Màntatas_97] R. López de Màntaras, J. Amat, F. Esteva, M. López, and C. Sierra, "Generation of Unknown Environment Maps by Co-operative low-cost Robots". In Proceedings of the First *International*

Conference on Autonomous Agents, Marina del Rey, February 5-8, 1997. ACM Press.

[López-Sánchez_97a] M. López-Sánchez, R. López de Màntaras, and C. Sierra, "Incremental Map Generation by low-cost robots based on possibility/necessity grids". In Proceedings of the Thirteenth *Annual Conference on Uncertainty in Artificial Intelligence*, Rhode Island USA, August 1-3, 1997, pp. 351-357.

[López-Sánchez_97b] M. López-Sánchez and R. López de Màntaras, "Incremental Map Generation using Behaviour-Based Autonomous Robots". *Jornades d'Intel.ligència Artificial de l'ACIA*, Universitat de Lleida, Spain September 1997. pp. 30-40.

[López-Sánchez_98a] M. López-Sánchez, G. Sukhatme, and G. A. Bekey, "Mapping an outdoor environment for path planning". Research Report from the University of Southern California (USA) Reference number: 98-683. Available at http://www.usc.edu/dept/cs/tech.html. June 1998.

[López-Sánchez_98b] M. López-Sánchez and R. López de Màntaras, "Refinement of environmental maps obtained from autonomous robot exploration", First *Catalan Conference on Artificial Intelligence* CCIA, Tarragona, Spain, October 1998,pp. 342-245.

[López-Sánchez_98c] M. López-Sánchez, R. López de Màntaras, C. Sierra; "Possibility theory-based environment modelling by means of behaviour-based autonomous robots". In Proceedings of the 13th *European Conference on Artificial Intelligence* ECAI, Brighton (UK), August 23-28, 1998, pp. 590-594.

[López-Sánchez_98d] M. López-Sánchez, F. Esteva, R. López de Màntaras, C. Sierra, and J. Amat, "Map Generation by Co-operative Low-cost Robots in Structured Unknown Environments". *Autonomous Robots Journal*, Kluwer Academic Publishers, Manufactured in The Netherlands, Vol. 5, pp. 53-61, 1998.

[López-Sánchez_99a] M. López-Sánchez, R. López de Màntaras, C. Sierra, "Map Generation by means of Autonomous Robots and Possibility Propagation Techniques". In Proceedings of the Third *International Conference on Autonomous Agents,* Seattle (USA), May 1-5, 1999. pp.380-381.

[López-Sánchez_99b] M. López-Sánchez, R. López de Màntaras, and C. Sierra, "Map generation by means of autonomous robots and possibility propagation techniques". Contribution to the book Fuzzy Logic techniques for autonomous vehicle navigation. Ed. Springer. Editors: Alessandro Saffiotti and Dimiter Driankov. In press.

[Lu_97] F. Lu and E. Milios, "Globally Consistent Range Scan Alignment for Environment Mapping". In *Autonomous Robots*, Vol 4, 1997, pp. 333-349.

[Mataric_90] M. J. Mataric. "A Distributed Model for Mobile Robot Environment-learning and Navigation". Master's Thesis. MIT, Cambridge, MA, 1990; also MIT, AI Lab, Technical Report AITR-1228.

[Mataric_94] M. J. Mataric, "Interaction and Intelligent Behaviour". Ph. D. Thesis. Also available as Research Report with number AI-TR 1495 at the Massachusetts Institute of Technology. August 1994.

[Matthews_88] P. Matthews, "Covering Problems for Brownian Motion on Spheres" in The Annals of Probability, 1988, Vol. 16, No. 1, pp 189-199.

[Moravec_85] [7] H. P. Moravec and A. Elfes "High Resol ution Maps from Wide Angle Sonar". In Proceedings of the *IEEE International Conferen ce on Robotics and Automation*, St Luis, 1985, pp. 116-121.

[Murao_97] H. Murao and S. Kitam ura, "Q-learning with Adaptive State Space Constr uction". In Proceedings of the Si xth *European Workshop on Learning Robots*, Brighton, UK. 1997. pp. 4-13.

[Nash-Williams _59] J. A. Nash-Williams, "Random walk and electric currents in networks " Proc. *Camb. Phil. Soc.* Vol.55, 1959, pp. 181-194.

[Nelson_89] R. C. Nelson, "Visual homing using an associati ve memory". In Proceedings of the *Image understanding Work shop*, 1989.

[Nilsson_69] N. J. Nilsson, "A Mobile Automation: an Application of AI Techniques ", in Proceedings of the *First International Joint Conferen ce on Artificial Intelligen ce*, San Francisco, 1969. pp. 509-520.

[Oriollo_98] G. Oriolo, G. Uli vi, and M. Vendittelli, "Real Time Map Building and Navigation for Autonomous Robots In Unknown Environments". In *IEEE Transactions on Systems, Man, and Cybernetics.* Vol. 2, no. 3, Part B, pp.316-333, 1998.

[Pagac_96] D. Pagac, E. M. Nebot, and H. D urrant-White. "An Evidential Approach to Probabilistic Map-B uilding". In Proceedings of the *IEEE: International Conferen ce on Robotics and Automation*. 1996. pp. 745-750.

[Parker_94] L. E. Parker, "Heterogeneous Multi-Robot Co-operation". Ph. D. Thesis. Also available as research report number AI-TR 1465 at the Massachusetts Institute of Technology. 1994.

[Payton_87] D. W. Payton, "An architecture for reflexive autonomous vehicle control" in Proceedings of the *International Conference on Robotics and Automation*, San Francisco, 1986, pp.1838-1845.

[Parker_94] J. R. Parker, "Practical Computer Vision Using C", *Willey Professional Computing*, Canada 1994.

[Pearl_84] J. Pearl, "Intelligent Search Strategies for Computer Problem Solving", *Addison-Wesley Publishing Company,* California, 1984.

[Poloni_95] M. Poloni, G. Ulivi, and M. Vendittelli, "Fuzzy Logic and Autonomous Vehicles: Experiments in Ultrasonic Vision", *Fuzzy Sets and Systems*. Vol. 69, 1995, pp15-27.

[Prescott_96] T. J. Prescott, "Spatial representation for navigation in animats" in *Adaptive Behaviour,* Vol. 4, n. 2, 1996.

[Rao_96] N. S. V. Rao, V. Protopopescu, and N. Manickam, "Co-operative Terrain Model Acquisition by a Team of Two or Three Point-Robots". In Proceedings of the *IEEE: International Conference on Robotics and Automation*. Minneapolis. April 1996, pp. 1427-1433.

[Rekleitis_97] I. M. Rekleitis, G. Dudek, and E. E. Milios, "Multi-Robot Exploration of an Unknown Environment, Efficiently Reducing the Odometry Error". In Proceedings of *International Joint Conference on Artificial Intelligence IJCAI.* Nagoya-Japan. August 1997. pp. 1340-1345.

[Russ_95] J. C. Russ, "The Image Processing Hand Book", *IEEE Press*, 1995.

[Russell_95] S. J. Russell and P. Norving, "Artificial Intelligence. A modern Approach", *Prentice Hall,* Edited by Russell and Norving, 1995. Chapter 25: Robotics, pp.773-811.

[Saffiotti_93] A. Saffiotti, E. H. Ruspini, and K. Konolige, "Integrating Reactivity and Goal Directedness in a Fuzzy Controller". In Proceedings of *the Second Fuzzy-IEEE Conference,* 1993. New York: IEEE.

[Saffiotti_95] A. Saffiotti, E. H. Ruspini, and K. Konolige, "A Multi-valued Logic Approach to Integrating Planning and Control" *Artificial Intelligence,* Vol. 76, n.1/2, 1995, pp 481-526.

[Saffiotti_96] A. Saffiotti and L. Wesley, "Perception-based Self-localisation Using Fuzzy Locations". In Dorst, Lambalgen, and Voorbraak, Eds. Reasoning with Uncertainty in Robotics. *Lecture Notes in Artificial Intelligence* 1093 (Springer, 1996). pp.368-385.

[Saffiotti_97] A. Saffiotti, "The Uses of Fuzzy Logic in Autonomous Robot Navigation". In *Soft Computing* Vol. 1, n. 4, 1997.

[Savage_72] L. J. Savage, "The foundations of Statistics", *Dover Publications*, 1972.

[Schultz_98] A. Schultz and W. Adams, "Continuous Localisation Using Evidence Grids". In Proceedings of the *IEEE International Conference on Robots and Automation.* Leuven-Belgium 1998.

[Sgouros_96] N. M. Sgouros, G. Papakonstantinou, P. Tsanakas, "Localised Qualitative Navigation for Indoor Environments". In Proceedings of the *IEEE: International Conference on Robotics and Automation ICAR.* 1996. pp 921-926.

[Shafer_76] G. Shafer, "A mathematical Theory of Evidence " *Princeton Univ. Press*, Princeton N. J., 1976.

[Shatkay_97] H. Shatkay and L. P. Kaelbling, "Learning Topological Maps with Weak Local odometric Information " In Proceedings of the Fifteenth *International Joint Conference on Artificial Intelligence IJCAI*, Japan, 1997, pp. 920-927.

[Shen_98] W. M. Shen, J. Adibi, R. Adobbati, B. Cho, A. Erdem, H. Moradi, B. Salemi, and S. Tejada, "Building Integrated Robots for Soccer Competition". In Proceedings of the third *International Conference on Multi-Agent Systems*. France. July 1998. pp. 465-466.

[Smets_88] P. Smets, "Belief Functions" In *Non-Standard Logics for Automated Reasoning.* Edited by P. Smets, A. Mamdani, D. Dubois, H. Prade. Academic Press, 1988. pp 235-277.

[Szepesvari_97] C. Szepesvari, Z. Kalmar, and A. Lorincz, "Module Based Reinforcement Learning for a Real Robot ". In Proceedings of the Sixth *European Workshop on Learning Robots*, Brighton, UK. 1997. pp. 22-31.

[Thorpe_90] C. Thorpe and J. Gowdy, "Annotated maps for autonomous land vehicles". In Proceedings of the *Image Understanding Workshop*, 1990.

[Thrun_97] S. Thrun, "To Know or Not to Know: On the Utility of Models in Mobile Robotics ", In *AI Magazine*, Vol. 18, n. 1, Spring 1997, pp. 47-54.

[Thrun_98a] S. Thrun, "Learning Metric-Topological Maps for Indoor Mobile Robot Navigation". In *Artificial Intelligence*, Vol. 99, 1998, pp. 21-71.

[Thrun_98b] S. Thrun, W. Burgard, and D. Fox "A Probabilistic Approach to Concurrent Mapping and Localisation for Mobile Robots". In *Machine Learning*, Vol. 31, 1998, pp. 29-53.

[Torra_95] V. Torra, "A new combination Function in Evidence Theory". In *International Journal of Intelligent Systems* (IJIS). Vol. 10, n.12, 1995. pp 1021-1033.

[Trillas_93] E. Trillas, "Fuzzy Sets and Fuzzy Logic" i*n Arbor: Lógica Fuzzy para Razonar como la gente*. Sep-Oct 1993. pp-83-106.

[Vandorpe_96] J. Vandorpe, H. Van Br ussel, and H. Xu. "Exact Dynamic Map Building for a Mobile Robot using Geometrical Primitives Produced by a 2D Range Finder". In Proceedings of the *IEEE: International Conference on Robotics and Automation*. 1996, pp 901-908.

[Veloso_98] M. Veloso, P. Stone, and K. Han, "The CMU-United-97" Robotic Soccer Team: Perception and M ultiagent Control". In Proceedings of the third international conference on *Autonomous Agents*. Minneapolis. 1998. pp. 78-85.

[Wagner_98] I. A. Wagner, M. Lindenba um, and A. M. Br uckstein, "Robotic Exploration, Brownian Motion and Electrical Resistance" in Proceedings of the 2nd International workshop on Randomisation and Approximation Techniq ues in Computer Science, Barcelona, Oct 98. pp. 116-130.

[Winston_92] P.H. Winston "Artificial Intelligence", *Addison-Wesley*, Reading, Massach usetts, third edition, 1992.

[Ya-Lun-Chou_72] Ya-Lun Chou. "Analisis Estadístico". *Muñoz Publications*. Mexico D F. 1972.

[Yamauchi_96] B. Yamauchi, "Mobile Robot Localisation in Dynamic Environments using Dead-reckoning and Evidence Grids ". In Proceedings of the *IEEE International Conference on Robotics and Automation*". Minneapolis 1996. pp. 1401-1406.

[Yamauchi_98] B. Yamauchi, "Frontier-Based Exploration Using Multiple Robots" in Proceedings of the Second International Conference on Autonomous Agents (Agents' 98). May 1998. Minneapolis, pp. 47-53.

[Yoshida_98] T. Yoshida, K. Hori, S. Nakasuka, “A Reinforcement Learning Approach to Cooperative Problem Solving”. In Proceedings of the *International Conference on Multi-Agent Systems.* Paris !998. pp.479-480.

[Zadeh_92] L. A. Zadeh and Kacprzyk. “Fuzzy Logic for the Management of Uncertainty”, *Wiley*. New York, 1992.

[Zipser_86] D. Zipser, “Biologically Plausible Models of Place Recognition and Goal Location”. In J. L. McClellant & D. E. Rumelhart (Eds.) *Parallel Distributed Processing: Explorations in the Micro-Structure of Cognition*. Cambridge, MA: Bradford. Vol. 2, 1986, pp. 432-470.

[Zimmer_96] U. R. Zimmer, “Robust World Modelling and Navigation in a Real World”. In *Neurocomputing*, Vol. 13, n. 2-4, 1996, pp. 247-260.

Index