Algorithms and Heuristics for Total and Partial Constraint Satisfaction

Javier Larrosa Bondia

Foreword by Pedro Meseguer

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
Pedro Meseguer
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Javier Larrosa Bondia
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

ISBN: 84–00–07743–1
Dip. Legal: B–42484–98

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.

Printed by CPDA-ETSEIB.
Avinguda Diagonal, 647.
08028 Barcelona, Spain.

A mis padres,
por transmitirme su confianza e ilusión

A Mari,
por su apoyo incondicional

Contents

Foreword

This book deals with an old topic in Constraint Satisfaction: the development of computational methods to solve constraint problems as efficiently as possible. Because of the intractability of constraint satisfaction, this is a long-term goal and this book contains some contributions to it. Total and Partial Constraint Satisfaction have been often considered as completely different problems. This work takes a common perspective on both problems, approach that has been shown very fruitful. Original ideas on constraint processing have been successfully developed for both problems, and some ideas already considered for total constraint satisfaction have been adapted into the partial case.

The reader will find here new algorithmic developments, original extensions of existing, and novel heuristic approaches which have been shown to be significatively more efficient than previous solving methods currently in practice. No assumption is taken over the target problems and only systematic search strategies are used. In this sense, the computational methods presented here are completely general and can be applied to any constraint satisfaction problem. Dealing with algorithms, a major issue is to assess their relative performance. When possible, performance improvements are given in terms of dominance of structural parameters. If not, empirical effort measures are presented. Regarding benchmarks, the binary random model is used, together with some other real problems such as crossword puzzles and job-shop scheduling.

Bellaterra, July 1998

Pedro Meseguer
Researcher of the IIIA-CSIC

Acknowledgements

Many people have helped me in this work either directly or indirectly. A good deal of the ideas presented in this Thesis were conceived within the IIIA/CSIC walls. I'm much obliged to all its people for letting me share the charming atmosphere that they have in the lab. In particular, I thank Francesc Esteva —IIIA head— for letting me use every facility in the laboratory.

I thank Cupid for not blindly shooting all his arrows and providing me with connections between LSI/UPC and IIIA/CSIC. Amalia Duch - Pablo Noriega and M. Luisa Bonet - Jordi Levy have been excellent messengers carrying many drafts of this Thesis from Barcelona to Bellaterra and viceversa. I'm not forgetting Carlos Sierra, another faithful messenger offering me an excellent 24 hour hand-to-hand courier service.

I'm full of gratitude to my teaching mates. German Rigau, Lluis Vila, Xavi Burgués and Ramon Sangüesa have given me their support any time that I have needed it.

I'm very thankful to Mark Stewart Turnham for doing his best in ironing the English. His revision was always done under pressure and he never had the last version of the document. Therefore, I take complete responsibility for every remaining typo (unless it is a Minnesota idiom).

I thank Thomas Schiex and Gerard Verfaillie. Chapter 4 would never have been as it is without a fruitful visit to INRA in Toulouse.

Most of this work has been partially funded by the Spanish CICYT under the projects #TAP93-0451 and TIC96-0721-C02-02.

Last but not least, I'm much obliged to Pedro Meseguer. I thank him for his wise advise, for not sending me to hell during the many discussions where I did not want to listen to him, and for cheering me up every time that I felt dispair. It's been a treat working with him.

Abstract

Many important problems arising in Artificial Intelligence and other fields of Computer Science can be naturally expressed as a CSP. The task of finding a CSP solution is called *total constraint satisfaction.* It may happen that a CSP has no solution. In that case, it is of interest to search for assignments that best respect constraints. This situation is often called *partial constraint satisfaction.* This Thesis is devoted to the development of efficient algorithms for total and partial constraint satisfaction.

The central idea of our work is to show that total and partial constraint satisfaction have many common features that can be successfully exploited for algorithmic development. In this sense, this Thesis takes a step forward in the development of algorithms for both total and partial constraint satisfaction inspired by common key ideas. In this Thesis we explore a number of directions and develop several algorithms which outperform state-of-the-art competitors. The main ideas that we have developed are:

Subproblem merging: we show that depth-first search performs redundant search when dealing with similar subproblems. We present a search space transformation which involves the fusion of similar subproblems. We develop algorithms substantiated on that idea for total and partial constraint satisfaction.

Combining search with local consistency: the enforcement of local consistency has been shown to be a fruitful approach for early dead-end detection in total constraint satisfaction. We develop the same idea for the partial constraint satisfaction case. We use local consistency information (DAC counts) to improve the branch and bound lower bound. We present different algorithms of increasing sophistication.

Lazy computation: different CSP algorithms perform more computation than strictly needed. In total constraint satisfaction, lazy computation techniques have been found to be useful in overcoming this problem. We extend this idea to partial constraint satisfaction and show that in this case it is even more suitable. We present a lazy algorithm which avoids performing many of the computations.

Heuristics: although not completely equivalent, heuristics for total and partial constraint satisfaction can be developed targetting the same goals. We present a unifying perspective of total and partial constraint satisfaction in terms of global optimization and use it as a source of inspiration for heuristic generation. Our heuristics, which are valid for both the decision and the optimization CSP problem, show to be competitive with both generic heuristics and specific job-shop techniques.

Chapter 1

Introduction

This Thesis deals with search algorithms for constraint satisfaction problems. A *constraint satisfaction problem* (CSP) consists of a set of *variables*; each variable has associated a finite set of possible *values* (its *domain*); and there is a set of *constraints* restricting the values that variables can simultaneously take. Solutions are assignments of values to variables respecting the problem constraints. The task of finding a CSP solution is called *total constraint satisfaction.* However, it may happen that a CSP has no solution. In that case it is of interest to search for assignments that best respect constraints. This situation is often called *partial constraint satisfaction.* Total and partial constraint satisfaction are *decision* and *optimization* problems, respectively. Our work is devoted to search algorithms for both total and partial constraint satisfaction.

Search has been a central topic of *Artificial Intelligence* because many cognition tasks can be naturally expressed as search problems. CSP are a particular kind of search problems, which have some particularities —such as having a search space representable by a depth-bounded tree or the pruning effect of propagating decisions— that CSP-specific search algorithms take advantage of. Besides, there are many real problems from very different domains can be expressed as CSP.

Total and partial constraint satisfaction have often been considered two different research topics focusing on different points. The main interest on total constraint satisfaction has been the development of computationally efficient algorithmic techniques for either general or specific classes of CSP. Regarding partial constraint satisfaction, most previous research was devoted to the development of frameworks for problem representation, without forgetting algorithmic contributions in the case that every constraint is considered equally important.

The central idea of our work is to show that *total and partial constraint satisfaction have many common features that can be successfully exploited for algorithmic development.* In this sense, all the contributions that we present in this work are either applicable to both total and partial constraint

satisfaction, or extend previous work on total constraint satisfaction to partial constraint satisfaction. The intent of this Thesis is to take a step forward in the development of algorithms for both classes of problems inspired by common key ideas.

The structure of this Chapter is as follows. In Section 1.1, we motivate the development of efficient algorithms for constraint satisfaction. In Section 1.2, we establish both the scope of our work and the orientation that we give to it. The contributions of our work are presented in Section 1.3. Finally, we overview this Thesis in Section 1.4.

1.1 Motivation

Constraint satisfaction is gaining a great deal of attention because many combinatorial problems arising in Artificial Intelligence and other areas of Computer Science can be expressed in a natural way as CSP. Practical applications of total and partial constraint satisfaction can be found in a variety of domains such as *scheduling* [Minton *et al.*, 92; Zweben and Fox, 94; Agnèse et. al, 95], *timetabling* [Frangouli *et. al.*, 95; Yoshikawa *et al.*, 96], *propositional reasoning* [Selman *et al.*, 96], *machine vision, VLSI circuit design*, etc. In the last years, constraint satisfaction has progressed significantly[1]. Because of this progress, constraint-based technology is rapidly gaining importance in industry. Several companies, such as *Renault* and *British Telecom*, have recently started to exploit this technology [Wallace, 96b]. A recent article in *Byte* chose constraint logic programming as the paradigm *likely to gain most in commercial significance over the next 5 years* [Pountain, 95].

Regarding computational complexity, total constraint satisfaction is NP-complete, and partial constraint satisfaction is NP-hard. Therefore, it is believed that all algorithms for these problems will present an exponential worst-case behaviour. In this situation, and considering the practical importance of constraint satisfaction, developing *efficient average-case algorithms* is of obvious interest. One has to accept the existence of perverse problem instances for which these algorithms are not suitable. However, as better algorithms are developed, larger and more difficult instances can be successfully considered. Recent advances in constraint satisfaction techniques are a good example of this claim. While simple backtracking algorithms are unable to solve *toy* problems such as the *n*-queens, more powerful algorithms developed in recent years have been successfully applied to a number of medium and large size real domains [Wallace, 96b].

---

[1]Regarding academic research, an annual International conference on constraint processing was established in 1995, and the specialized scientific journal Constraints was launched during year 1996.

In this document, we present a number of enhancements to algorithms for total and partial constraint satisfaction. Because of the computational intractability of the problem, all of them have exponential time requirements. However, we show that our algorithms outperform state of the art competitors for sufficiently large and interesting classes of problems. In that sense, our work contributes to the development of new algorithms of increasing efficiency which will eventually allow for the applicability of constraint satisfaction techniques to a broader spectrum of problems.

To date, most research on constraint techniques has been devoted to total constraint satisfaction, while partial constraint satisfaction was considered a secondary goal. This Thesis attempts to partially balance this situation. It is known that general schemas for optimization, such as branch and bound, can solve the partial constraint satisfaction problem. However, we believe that these general schemas can be greatly improved by developing CSP-specific techniques and integrating them into the optimization algorithms. In this work, we give evidence of this claim.

In many real constraint problems, solutions have direct economical impact. Consequently, there often exists an implicit preference criterion among solutions that can be made explicit by adding new constraints. For example, consider an airline crew scheduling problem where the task is the crew assignment for a set of flights (the same person may work on two different crews, provided their flight do not overlap). Assuming the company has the necessary staff to fulfil the basic needs, it is of obvious interest to add new constraints such as restricting the maximum time that a person must wait between two consecutive flights. In that kind of situation, the initial decision problem may become overconstrained and its optimization counterpart becomes of interest. In this work, we emphasize that partial constraints satisfaction complements total constraint satisfaction. Therefore, it is of practical interest to develop efficient algorithms for both tasks.

1.2 Scope and Orientation

The boundaries of this work are established by the following decisions:

1. *Practical constraint solving*: The final objective of our work is to contribute to the development of algorithms that can actually be applied in real domains. In that sense, every idea that we explore has immediate algorithmic implications that we motivate and develop. Thus, the end-product of our contributions are specific algorithms that have been implemented and can be tested on any CSP.
2. *General constraint solving*: We have mentioned that constraint satisfaction is computationally untractable. One way to circumvent

this intrinsic drawback is to characterize classes of problems that can be efficiently solved. However, our research does not fall into this line of work. We do not make any assumption about the problems that we attempt to solve. In practice, it means that our algorithms consider a CSP in its explicit form, where the only permitted operation is to ask about the consistency of a potential assignment. Therefore, they cannot take advantage of the problem semantics. For this reason, our methods are motivated in a general-purpose context and are expected to be applicable to a broad spectrum of domains. Some evidence of this claim is given in Chapter 7, where it is shown that our generic methods can be effectively applied in the job-shop problem.

3. *Systematic constraint solving*: A different approach to circumvent the computationally intractability of constraint satisfaction uses incomplete search schemas (sub-optimal for partial constraint satisfaction). These algorithms typically attempt to solve problems using local optimization techniques enhanced with some stochasticity to break out from local optima. They have been found very useful for some domains. However, in our work we do not consider this approach. All our algorithms explore the space of solutions in a systematic manner. Quoting [Pearl, 85], we are concerned with algorithms with the two following properties:
   (a). Do not leave any stone unturned.
   (b). Do not turn any stone more than once.
   Because of property (a) our algorithms are *complete* (*optimal*, in the partial constraint satisfaction case).
4. *Empirical evaluation*: Because of the practical orientation of our work and the recognized exponential worst-case behaviour of our algorithms, the assessment of our contributions is mainly supported by empirical methods. On occasions, we have been able to prove the superiority of our algorithms by purely theoretical methods showing that one algorithm cannot perform worse than another. However, even in those cases, the detected superiority could not be theoretically quantified. In general, each of our contributions is experimentally evaluated. In our experiments, we mainly use random binary problems because they have become the most widely used benchmark in the CSP community. Random problems have nice properties to benchmark algorithms (Section 4.2.4). However, we are also aware of their limitations, especially when exporting conclusions obtained on random problems to other domains.
5. *Binary constraints*: In this work, we have decided to restrict our attention to binary problems (namely, all constraints restrict pairs of variables). There are different reasons for this decision:
   (a). Most previous research on algorithms assumes binary problems. Thus, most existing algorithms are described under this assumption.

(*b*). Algorithmic insights are generally easier to understand for binary than for *n*-ary problems.

(*c*). Any CSP can be transformed into an equivalent binary CSP [Rossi et al., 90]. Therefore, binary CSP are, in a sense, representative of all CSP. However, it has to be mentioned that transforming a non-binary into a binary CSP produces a significant increase in the problem size, so the transformation may not be practical.

(*d*). Many problems of interest are directly expressed as binary CSP. Hence, the class of binary CSP is important by itself.

6. *Maximal constraint satisfaction*: There are many ways in which the optimization criterion for partial constraint satisfaction can be defined. Different frameworks associate constraints with specific semantics in terms of priorities [Schiex, 92], preferences degrees [Martin-Clouaire, 92], probabilities [Fargier and Lang, 93], or in terms of algebraic operators [Schiex *et al*, 95; Bistarelli *et al*, 95]. In our work we identify partial constraint satisfaction with MAX-CSP (*i.e.*: to maximize the number of satisfied constraints). Our choice is substantiated in the following facts:

(*a*). MAX-CSP is a *simple* model under which the intuition of algorithms is easily understood.

(*b*). MAX-CSP is a *general* model that does not require constraints to have special semantics. For this reason, algorithms for MAX-CSP are more likely to be applicable to other frameworks (some evidence of this claim is presented in the Appendix, where we show how MAX-CSP techniques are extended to fuzzy CSP, where constraints are fuzzy relations).

(*c*). MAX-CSP allows the use of the *same* definition of the problem for total and partial constraint satisfaction. This is especially useful in our approach where we attempt to exploit common features for total and partial constraint satisfaction algorithms.

1.3 Contributions

No matter if they are under or overcostrained, constraint satisfaction problems have common features that algorithms can take advantage of. For this reason, many techniques that exploit CSP-specific features can be used (or adapted) for the different constraint satisfaction tasks. With this idea in mind, we present total and partial constraint satisfaction and show how the same general techniques can be fruitfully adapted to both problems.

To introduce the contributions of our work we anticipate that we are concerned with backtracking-based algorithms. These algorithms do a depth-first traversal on a search tree such that each tree level corresponds to a problem variable and different tree nodes correspond to different

assignment alternatives. Figure 1.1 shows the search tree corresponding to a problem with three variables and three values per variable. The exponential worst-case behaviour of these algorithms comes from the exponential size of the search tree. In this context, techniques aiming at an efficient tree traversal are of clear interest. In the following we briefly describe the contributions of our work.

1.3.1 Merging Similar Subproblems

The first idea we explore is how we can transform the search space to make it more suitable for a given problem. It may happen that different values of the same variable have a similar constraining behaviour. In that case, algorithms that traverse the standard search tree are not appropriate because they do not take advantage of value similarity. They consider the similar values as completely different and solve their associated subproblems independently. As a result, the algorithms duplicate part of their work.

We show that one efficient way to deal with this situations is to transform the search space by merging sibling subtrees corresponding to similar values. In our approach, we temporarily assign more than one value to the current variable. As a result, we obtain narrower and higher trees. Figure 1.2 depicts the resulting tree that we obtain with our approach if we take the three variable problem and merge its two first-level subtrees into one. Observe that pairs of nodes in Figure 1.1 are merged into a single node in Figure 1.2. The cost is the addition of an extra level where the values responsible for the merging are finally disambiguated. We show that merging subtrees associated with similar values can significantly reduce the cost of the search. This idea can be seen as an extention of previous approaches with the same underlying intuition, in the sense that it is useful for a broader spectrum of situations.

Our approach is suitable for both total and partial constraint satisfaction. We develop two algorithms, one for each case, that dynamically merge subtrees when sufficiently similar values are detected and empirically show that they can greatly increase search efficiency.

1.3.2 Exploiting Local Consistency Information in Partial Constraint Satisfaction

We say that a node is in a dead-end when the subtree below it does not contain any problem solution (even a temporary solution in the partial constraint satisfaction case). When an algorithm falls into a dead-end, it is condemned to unsuccessfully traverse the corresponding subtree. Thus, visiting dead-end nodes is a source of inefficiency. There are certain

situations where an algorithm can detect that it is visiting a dead-end node. Then it can abandon this line of search and backtrack to the node's parent. It is of obvious interest to provide algorithms with powerful dead-end detection capabilities so they can backtrack soon after falling into dead-ends. The earlier dead-ends are detected, the fewer nodes the algorithm has to visit to solve a problem.

In the total constraint satisfaction context, techniques for dead-end detection involve combining search with local consistency. The idea is to achieve some level of local consistency at each search state because during local consistency enforcement the dead-end may be detected. In addition, if no dead-end is detected, local consistency removes values which will never appear in a solution, so it produces a smaller subproblem. The process of enforcing local consistency is usually called the propagation of the current assignment.

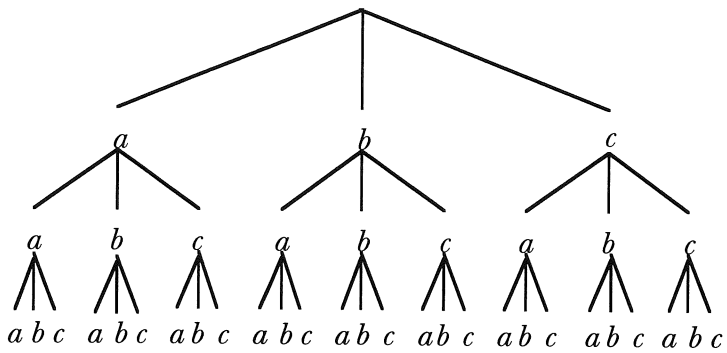


Figure 1.1: Search tree of a problem with three variables and three values per variable.

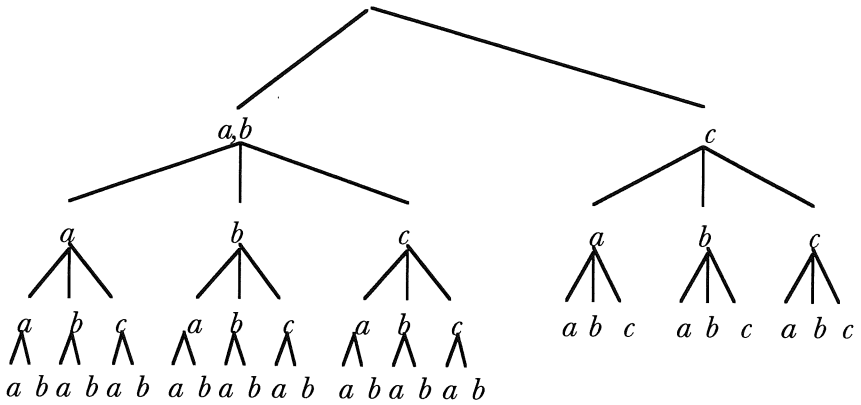


Figure 1.2: A search tree transformation produced by merging two subtrees.

We have explored the suitability of adapting the same idea in the partial constraint satisfaction context. We have found that, if algorithms are enhanced with the use of local consistency information gathered during a pre-processing step, their performance can be greatly improved. In addition, we have found that the use of local consistency information makes apparent a complexity peak in random problems not reported before. This complexity peak is unnoticeable when no local consistency information is used.

We present both theoretical and experimental evidence of the significance that this approach has in algorithmic efficiency. Furthermore, we have developed a sequence of incrementally more sophisticated algorithms which outperform previous ones by several orders of magnitude. Today, our algorithms can be considered among the most effective for partial constraint satisfaction.

The underlying ideas of these algorithms can be extended to more general frameworks of partial constraint satisfaction. To support this claim, we have developed and tested some of them for fuzzy constraint satisfaction problems (FCSP). FCSP extend MAX-CSP in that it allows the use of constraints with intermediate satisfaction degrees.

1.3.3 Lazy Evaluation

It is a matter of fact that enforcing local consistency during search is an efficient approach for early dead-end detection. In general, enforcing higher levels of consistency produces better dead-end detection. However, achieving higher levels of consistency requires more computational effort. Thus, there is a trade-off between the cost of propagation and the gains that come with it. In this context, it is of practical importance to develop efficient propagation methods.

Lazy evaluation is a general algorithmic technique which consists in delaying computation until it is strictly necessary. As a result, no redundant computations are done at the extra cost of more complex algorithms. In the total constraint satisfaction context, lazy evaluation has been successfully applied to propagate the effect of the current assignment. Using lazy evaluation, the same effect can be obtained avoiding some redundant computation.

We have explored the applicability of lazy evaluation in partial constraint satisfaction and have found that this approach is even more appropriate in this context. The reason is that in algorithms for partial constraint satisfaction there are more sources of redundant computation. We have developed a lazy algorithm which avoids performing a good deal of computation to evaluate the dead-end condition at each visited node. We have shown that this approach can be naturally combined with other

algorithms presented in this work. Thus, their efficiency can be joined. Our experiments give support to the suitability of the approach.

1.3.4 Heuristic Search Guidance

It has been known for a long time that the order in which variables and values are considered in depth-first search has a profound impact in algorithm efficiency. For this reason, variable and value ordering heuristics have been a pervasive issue in constraint satisfaction (especially in the total satisfaction case). Most heuristics for variable selection can be explained in terms of the fail-first principle, which states that variables should be selected aiming to drive search to a failure. Most value ordering heuristics can be explained in terms of the succeed-first principle, which affirms that values should be selected aiming to drive search to success.

We present a new perspective of constraint satisfaction and show its usefulness for heuristic generation. Our approach is based on the analysis of the labelling problem, a formalism arising in the field of computer vision that is closely related to constraint satisfaction. From this analysis, we extract that total and partial constraint satisfaction can be seen as the global optimization of the so-called average local consistency function. We use this point of view as a source of inspiration for variable and value ordering heuristics. More precisely, we propose the use of gradients to implement the fail-first and the succeed-first principles. Regarding variable ordering, we introduce the lowest support heuristic, which selects variables corresponding to low gradient subspaces. Regarding value ordering, we introduce the highest support heuristic, which selects values corresponding to high gradient directions. Since computing gradients is expensive, we present an approximate version of the heuristics which avoids the computation of exact gradients.

Comparing our heuristics with other constraint satisfaction approaches, we believe that our heuristics are more general because they have deeper foundations (they are based on gradients, which have a well known topological interpretation), include variable and value orderings in the same framework, and can be applied to both total and partial constraint satisfaction.

The suitability of our heuristics is evaluated in the job-shop problem. We show that our heuristics, which are developed in a general purpose context, are competitive with specific methods for the job-shop. The interest of our approach is that it does not include any domain-dependent element or any parameter that has to be adjusted manually for this particular domain. In that sense, our generic approach is more robust and more applicable to other problem instances than specific approaches. Our claim is based on experimental results on a classical job-shop benchmark that

has attracted a considerable amount of research in the job-shop community from an AI perspective.

1.4 Overview

This thesis is structured in eight Chapters and one Appendix. In Chapter 2, we revise previous work on constraint satisfaction. It contains basic terminology and some algorithms that will be subsequently used in our work. Furthermore, it presents some relevant results in the area, especially those that are related to our work. Regarding total constraint satisfaction, it covers depth-first search, local consistency, look-ahead and look-back algorithms, heuristics, and combined approaches. Regarding partial constraint satisfaction, it covers depth-first branch and bound, look-ahead algorithms and heuristics. Next, we address the topic of algorithmic evaluation and discuss some different approaches and the suitability of using random problems to benchmark algorithms. Finally, we give a historical account of the research advances that have occurred in parallel to our work.

Chapter 3 is devoted to subproblem merging. After reviewing previous work on the subject, we motivate and develop our approach in general constraint satisfaction terms. Next, we develop FCw, an algorithm for total constraint satisfaction with merging capabilities and analyze its behaviour. Then we develop PFCw, its partial constraint satisfaction counterpart. Finally, we evaluate their performance using random problems and crossword puzzles.

Chapter 4 is devoted to the combination of local consistency with search in the partial constraint satisfaction context. After reviewing previous work, we present theoretical results which show that the use of local consistency is a major advance over previous algorithms. Then, we present a set of algorithmic improvements which produce three new algorithms (PFC-GDAC, PFC-RDAC and PFC-MRDAC) of increasing sophistication. Finally, we experimentally evaluate our work. The experimental evaluation has two different parts. First, we examine and analyze the complexity peak of MAX-CSP that becomes apparent with local consistency enhanced algorithms. Second, we show that our new algorithms clearly outperform state of the art competitors on random problems.

Chapter 5 is devoted to the applicability of lazy evaluation in the partial constraint satisfaction context. After presenting previous work in the total constraint satisfaction context, we show the suitability of extending this approach to partial constraint satisfaction. We present a lazy algorithm, PLFC, and prove that it never performs worse that its non-lazy counterpart. Next, we show that lazy evaluation can be easily combined

with the algorithms presented in Chapter 4 and evaluate the suitability of their combination on random problems.

Chapter 6 is devoted to variable and value ordering heuristics for total and partial constraint satisfaction. In the first part of the Chapter we introduce the labelling problem and analyze its common points with constraint satisfaction. As a result, we obtain a unifying view of total and partial constraint satisfaction in terms of global optimization. In the second part of the Chapter, we use this optimization perspective of constraint satisfaction to devise effective variable and value ordering heuristics. More precisely, we use gradients to guide search. The suitability of our approach is assessed using random problems.

Chapter 7 presents a case-study of the applicability of the heuristics described in Chapter 6 to the job-shop problem. In this Chapter, we show that our heuristics, developed under general purpose motivation, are competitive with domain-specific techniques. Our claim is supported on the experimental results obtained on a classical job-shop benchmark.

Chapter 8 gives the conclusions of our work and proposes some lines of future work.

In the Appendix, we show how some of the algorithms described in Chapter 4 can be extended to fuzzy constraint satisfaction problems, a more general framework for partial constraint satisfaction than MAX-CSP.

The Chapters are quite self-contained. Thus, the reader can feel free to attempt to read this thesis in any desired order. There are two obvious exceptions: Chapter 2 has to be read first and Chapter 7 must be read after Chapter 6. Just one hint: if you are systematic when reading it (*i.e.*: you do not leave *any page* unturned) it is less likely that you will get trapped in a dead-end and it may save you some re-start. However, a nonsystematic reading may be fun, too.

Chapter 2

Related Work

Constraint Satisfaction is a research topic including many combinatorial problems arising in Artificial Intelligence and other areas of Computer Science. Its eminent practical applicability has attracted the interest of the research community. Currently, it is a very dynamic area of research that has produced significant advances in the last years. The goal of this Chapter is to overview part of this progress, especially in those lines of research that are close to our work. We do not attempt to be exhaustive, since there is so much to cover that a complete summary would require a dedicated book. However, we consider that it is sufficient to give a comprehensive introduction to the state-of-the-art in algorithmic aspects of constraint satisfaction.

After a brief introduction of basic concepts and notation, the Chapter is divided into two main parts. The first part is devoted to total constraint satisfaction and covers local consistency, depth-first algorithms and heuristics. It also gives a brief summary of results on tractable classes of CSP. The second part is devoted to partial constraint satisfaction and covers depth-first branch and bound and heuristics. In addition, we present an overview on the main issues related with algorithms evaluation. Finally, we give a historical account of what has been done in parallel to our work.

2.1 Preliminaries

A binary CSP is defined by a set of variables, $\{X_i\}$, each one taking values on its associated finite domain $\{D_i\}$, and a set of binary constraints[1] $\{R_{ij}\}$. The number of variables is n, the number of constraints is e, and we assume, without loss of generality, a common domain D for all variables, being m its cardinality. However, we will still use D_i when referring to

[1] We assume that problems are node-consistent, so no unary constraints are required in the formulation.

the domain of X_i in order to emphasize their relation. In general, $i,j,k,...$ will be indexes corresponding to variables, and $a,b,c,...$ will denote values. A constraint R_{ij} is a subset of $D_i \times D_j$, containing the permitted values for X_i and X_j. We associate a *constraint graph* with each binary CSP in which vertices represent variables and arcs connect pairs of constrained variables. Thus, we can talk about graph concepts associated with a CSP, such as a variable *degree* or a problem *connectivity*.

We will use the arrow, "$\leftarrow$", to denote the individual assignment of a variable (namely, $X_i \leftarrow a$ means that value a is assigned to variable X_i). In general, v^i will denote the value assigned to X_i. An *assignment* of values to variables is a set of individual assignments, $\{X_i \leftarrow v^i\}$, where no variable occurs more than once. An assignment can be either *partial*, if it includes a proper subset of the variables, or *total*, if it includes every variable. We say that an assignment is *consistent* if it does not violate any constraint. A *solution* to a CSP is a total consistent assignment. The task of finding a solution to a CSP or proving that it does not have any is usually referred as the task of *achieving total consistency*, or simply *solving* the CSP.

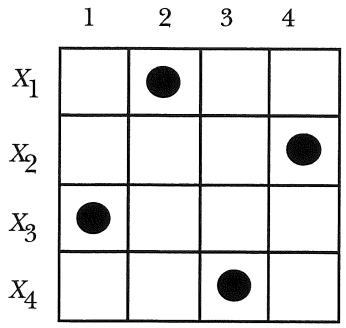
When a problem is overconstrained, *partial constraint satisfaction* is usually of interest. In this work, we associate partial constraint satisfaction with the problem of finding a total assignment that satisfies the maximum number of constraints. This problem is usually referred to as the *maximal constraint satisfaction problem* (MAX-CSP). Observe that CSP and MAX-CSP are the decision and optimization versions of the same problem.

Example 2.1:

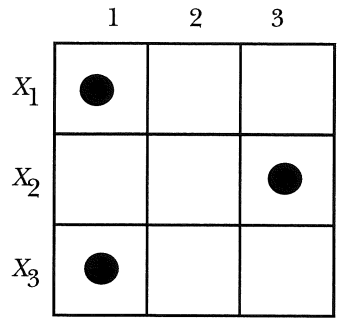
The n-queens problem has been typically used to illustrate CSP. The problem consist on placing n queens on an $n \times n$ chess board in such a way that no two queens attack each other. The n-queens problem can be represented as a binary CSP where each variable is associated with a board row, and its assignment denotes the position where the queen is placed. Thus, $\{X_1,..., X_n\}$ is the set of variables, and $\{1,..., n\}$ their common domain. Constraints restrict the valid positions for pairs of queens: Two queens cannot be placed in the same column, nor in the same diagonal (observe that the problem imposition that two queens cannot be placed in the same row is already guaranteed by the representation). Thus, an arbitrary constraint is of the form $R_{ij} = \{(a,b): a \neq b \text{ and } |i-j| \neq |a-b|\}$. In this particular domain there is a constraint between every pair of variables. Therefore, the constraint graph associated with the n-queens problem is a n-vertices clique.

Consider the 4-queens problem. The following board configuration represents a problem solution (black dots denote queens), which corresponds to the total assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 4, X_3 \leftarrow 1, X_4 \leftarrow 3\}$. It

is easy to see that it is a solution because it does not violate any constraint.



Consider now the 3-queens problem. It is easy to see that it does not have any solution (this problem is over-constrained). It may be of interest to find its *best* total assignment. For this problem, there are several total assignments that violate one constraint. The following picture represents one of them, associated with the assignment $\{X_1 \leftarrow 1, X_2 \leftarrow 3, X_3 \leftarrow 1\}$. It violates constraint R_{13} only.



Observation 2.1:

1. Solving a CSP is an NP-complete problem.
2. MAX-CSP is NP-hard.
3. From (1) and (2), it is believed that all of the algorithms for these problems will present exponential worst-case behaviour.

Proof:

1. It is obvious that CSP is an NP problem because we can check whether a total assignment is a solution or not in polynomial time. To show that it is NP-complete, observe that the graph colouring problem, which is NP-complete [Garey and Johnson, 79], can be

directly expressed as a CSP. Therefore, solving the CSP is NP-complete. More details on CSP complexity can be found in [Haralick and Shapiro, 79].

2. It was shown that CSP is NP-complete. It is clear that MAX-CSP reduces CSP because we can always find a CSP solution by finding its best solution and checking whether its distance is zero. Therefore, MAX-CSP is NP-hard.

The space of possible solutions for a CSP is the set of all total assignments. This set of assignments can be generated with the following procedure based on an ordering among variables and domain values. Consider the first variable and its m possible values; for each value, consider the second variable and its m possible values (it produces m^2 possibilities); for each combination of values, consider the third variable and its m possible values (it produces m^3 possibilities); an so on. If the process is done for the n problem variables, it generates a tree such that its leaves form the set of all possible total assignments in the problem. Figure 2.1 shows this tree for a problem with 3-variables and 3-values. Observe that each internal node corresponds to a partial assignment. Following any path from the tree root to a leaf, each step extends the partial assignment (initially empty) including one more variable. At each internal node, its associated partial assignment represents some decisions that have been already taken about part of the variables. Thus, the internal nodes are CSP subproblems where only the remaining variables have to be assigned. In this context, solving a CSP consists in finding a consistent tree leaf, and solving MAX-CSP consist in finding a leaf satisfying as many constraints as possible. It is obvious that CSP algorithms can be defined in terms of systematic tree traversals.

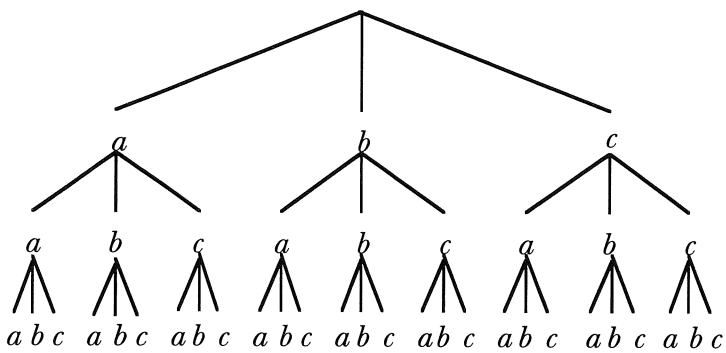


Figure 2.1: Search tree for a 3-variable 3-values per variable problem.

2.2 Algorithms for Total Constraint Satisfaction

2.2.1 Depth-first Algorithms

Depth-first is a general search technique frequently used because of its polynomial memory requirements. In the CSP context, depth-first search is particularly adequate because the search space can be naturally expressed as a tree with bounded depth (the number of variables), so there is no danger to get lost in an infinite branch. These algorithms traverse the search tree depth-first, where each visited node is associated with a *consistent* partial assignment. They search for a solution by continuously trying the extension of the current consistent partial assignment into a total one. At each node an unassigned variable is selected and its values are sequentially assigned. We say that an algorithm is in a *dead-end* when it visits a node that does not have any solution below. A dead-end is detected at nodes where all values are rejected as candidates for the assignment extension. When a dead-end is detected, the algorithm backtracks to previous nodes. During the search, the node that is being visited is called the *current node*. Assigned and unassigned variables are called *past* (**P**) and *future* (**F**) variables, respectively. The variable that is being assigned at the current node is called the *current variable*, and the value that is being tried at a given time is called the *current value*. Abusing notation, we will take **P** and **F** as the set of variables or the set of indexes, depending on the context.

Figure 2.2 shows *chronological backtracking* (BT), the simplest algorithm for CSP solving. P and F denote the sets of past and future variables, respectively. *Assg* is the current partial assignment and *Dom* is the set of future domains. After an initial call $BT(\{X_1,...,X_n\},\varnothing,\varnothing,\{D_1,...,D_n\})$, the algorithm returns *true* if the problem is solvable and *false* otherwise. If the problem is found to be solvable, the solution is recorded in a global variable *Sol*. If the set of future variables is empty, *Assg* is a problem solution, so it is recorded (line 2), and the search is abandoned. Otherwise, a variable is selected (line 5) and the algorithm sequentially attempts the assignment of its values. If the current value is consistent with respect to past assignments, a recursive call is made (line 12), so the search continues below the node. Otherwise, the current assignment is changed.

2.2.2 Consistency Enforcement Algorithms

Since detecting when a given problem is solvable is an NP-complete task, related but weaker properties about the problem may be useful. *Local consistency* is a family of increasingly harder properties about the problem [Montanari, 74; Mackworth, 77; Freuder, 78]. For example, the most basic concept of local consistency is *arc-consistency*. It ensures that any value in the domain of a variable has at least one consistent value among the domain of any other variable. *Path-consistency* ensures that given a consistent assignment involving two variables we can take any additional variable and consistently extend the assignment to the third variable. In general, *i-consistency* [Freuder, 78] ensures that any consistent assignment involving i-1 variables is extensible to include any additional variable forming a consistent assignment involving i variables. Thus, arc-consistency is 2-consistency, and path-consistency is 3-consistency. *i-strong consistency* [Freuder, 82] ensures that a problem is *k-consistent* for all $k \leq i$.

```
function BT (P, F, Assg, Dom) returns boolean
1       if (F = ∅) then
2               Sol := Assg
3               return (true)
4       endif
5       (Xi,Di):= select_current_variable_and_domain(F, Dom)
6       stop:= false
7       while (Di ≠∅ and not stop) do
8               a:= select_current_value(Di)
9               Di:= Di-{a}
10              NAssg:= Assg ∪ {Xi←a}
11              if (look_back(Xi, a, Assg)) then
12                      stop:=BT(P∪{Xi}, F-{Xi}, NAssg, Dom-{Di})
13              endif
14      endwhile
15      return (stop)
endfunction
function look_back(Xi, a, Assg) returns boolean
16      for all (Xj←v^j)∈Assg do
17              if (inconsistent(Xi←a, Xj←v^j) then return (false) endif
18      endfor
19      return (true)
endfunction
```

Figure 2.2: Chronological Backtracking.

Local consistency enforcement algorithms are polynomial algorithms that transform a given problem into an equivalent but more explicit problem by detecting constraints which are implicit in the problem and adding them explicitly. For example, arc-consistency algorithms transform a problem into an equivalent arc-consistent one by removing arc-inconsistent values. Path-consistency algorithms transform a

problem into an equivalent path-consistent one by adding binary constraints disallowing pairs of values that cannot be extended to consistent assignments of three variables. In general, i-consistency algorithms transform a problem into an equivalent i-consistent one by adding new constraints of arity i-1.

Using local consistency algorithms has two potential benefits:

- *Problem simplification:* The transformed problem has additional explicit information. Typically, a search algorithm takes advantage of it, and improves its efficiency. Nevertheless, in some cases achieving local consistency may degrade the subsequent search procedure [Prosser, 93b; Sabin and Freuder, 94].
- *Unsolvability detection:* during their execution, local consistency algorithms may detect that a problem is unsolvable.

The complexity of enforcing i-consistency is exponential in i [Cooper, 89]. Considering this high cost, there is a trade-off between the effort spent in pre-processing and the saving that it may produce.

Regarding binary CSP, arc-consistency —or weaker forms of arc-consistency— are commonly used to detect and remove unfeasible values before and during search. They are of interest because they have low time and space requirements. Path-consistency is not so useful in practice because it adds constraints to the problem so its structure is changed. Moreover, its space and time requirements are higher. Higher levels of local consistency are seldom used because of their prohibitive cost and because they add constraints of arity greater than two, transforming the problem into a non-binary one.

Many algorithms for arc-consistency enforcement have been presented. Starting from Mackworth's AC-1, AC-2 and AC-3 [Mackworth, 77], a series of increasingly more elaborated algorithms includes: AC-4 [Mohr and Henderson, 86], AC-6 [Bessière, 94], AC-7 [Bessière et al., 95].

AC-4 is a well known arc-consistency algorithm. Its worst-case time requirements are in $O(e \cdot m^2)$ which has been shown to be optimum. This algorithm is based on the concept of *support*. When two values are permitted by a constraint, we say that they support each other. Arc-consistency is held by those problems in which each value of a variable has at least one supporting value in the domain of each other variable. When one value does not have any support at some variable, it can be removed because it cannot belong to any problem solution. AC-4 makes this concept of support evident by using the following data structures:

- *CSupport* is an array of counters recording the number of supports that each value of a variable has from another variable. Thus, $CSupport_{iaj}$ is the number of supporting values that value $a \in D_i$ has in D_j.
- *LSupport* is an array of sets which records individual supports. Thus $LSupport_{ia}$ is the set of pairs (X_j, b) such that $a \in D_i$ gives support to them.

AC-4 works in two steps. The first step involves the initialization of the *CSupport* and *LSupport* data structures. During the initialization, values not having support at some variable are pruned from their domain and kept in *Pruned,* a set of pruned values. The second step involves the propagation of the effect of pruned values because their pruning may decrease some other values support. For each pruned value $a \in D_i$, its set of support is used to access those values it was giving support to. Then, its counter of supports from X_i is decremented. If it becomes zero, it means that these values do not have support from X_i anymore, so they are pruned and added to the set of pruned values that still remain to be propagated. AC-4 is given in Figure 2.3. Given a CSP, arc-consistency is achieved with the following two imbricated function calls:

```
(CSupport, New_Domains) := AC-4(ini_structures ({D1, ...,Dn}))
```

If some domain is empty after the arc-consistency enforcement, then the problem is unsolvable. Otherwise, the arc-consistent problem is the original problem subject to the new domains.

Example 2.2:

Consider the task of enforcing arc-consistency to the 3-queens problem. If we use AC-4, after the initialization step, the following support counters are obtained:

	1	2	3
X_1	$CSupport_{112}=1$ $CSupport_{113}=1$	$CSupport_{122}=0$ $CSupport_{123}=2$	$CSupport_{132}=1$ $CSupport_{133}=1$
X_2	$CSupport_{211}=1$ $CSupport_{213}=1$	$CSupport_{221}=0$ $CSupport_{223}=0$	$CSupport_{231}=1$ $CSupport_{233}=1$
X_3	$CSupport_{311}=1$ $CSupport_{312}=1$	$CSupport_{321}=2$ $CSupport_{322}=0$	$CSupport_{331}=1$ $CSupport_{332}=1$

From this information, we know that values $2 \in D_1$, $2 \in D_2$ and $2 \in D_3$ are unfeasible because all of them have some variable from which they do not obtain any support. These values are removed and their

elimination is propagated. Thus, propagating the deletion of $2 \in D_1$, $CSupport_{311}$ is decremented and it becomes zero. It means that its only supporting value in X_1 has been pruned, so $1 \in D_3$ becomes arc-inconsistent and is removed, too. The same thing occurs with $3 \in D_3$ which is also removed. Then, as a result of propagating the deletion of $2 \in D_1$, D_3 becomes empty. It shows that the 3-queens problem is unsolvable.

A good deal of work has been done on the detection of tractable classes of CSP. Since the exponential cost of depth-first search is associated with the number of nodes that it requires to visit (which is exponential in the worst-case), it is of interest to establish conditions under which the number of backtrackings can be bounded. The best case occurs when depth-first solves a problem in a backtrack-free manner. [Freuder, 82] shows that local consistency can be used to establish sufficient condition for a backtrack-free search.

Definition 2.1:

Given an arbitrary CSP and an ordering $\{X_1,..., X_n\}$ among its variables, a depth-first search in the CSP is *backtrack-free* under the ordering if given a consistent assignment $\{X_1 \leftarrow v^1,..., X_{i-1} \leftarrow v^{i-1}\}$ one can always find a value for X_i which is consistent with the assignment.

The level of local consistency required to guarantee backtrack-free search for a given CSP depends on its graph *width*.

Definition 2.2:

1. Given a graph and a total ordering among its nodes, the *width of a node* is the number of adjacent nodes that it has before, with respect to the ordering.
2. The *width of a graph under an ordering* is the maximum width of all the nodes in the graph under that ordering.
3. The *width of a graph* is the minimum width of the graph under all possible orderings among its nodes.

The following theorem relates the constraint graph topology with the level of local consistency required to guarantee backtrack-free search.

```
function ini_structures (Dom)
1       CSupport_iaj := 0  for all i,a,j
2       LSupport_ia := ∅ for all i,a
3       forall R_ij do
4               forall (a,b)∈R_ij do
5                       CSupport_iaj := CSupport_iaj +1
6                       CSupport_jbi := CSupport_jbi +1
7                       LSupport_ia := LSupport_ia ∪{(X_j,b)}
8                       LSupport_jb := LSupport_jb ∪{(X_i,a)}
9               endfor
10      endfor
11      Pruned := ∅
12      stop:= false
13      forall R_ij while ( not stop) do
14              for a ∈D_i if(CSupport_iaj  = 0) do
15                      Pruned := Pruned ∪{(X_i,a)}
16                      D_i:= D_i-{a}
17              endfor
18              for b ∈D_j if(CSupport_jbi  = 0) do
19                      Pruned := Pruned ∪{(X_j,b)}
20                      D_j:= D_j-{b}
21              endfor
22              if (D_i = ∅or D_j = ∅) then stop:= true endif
23      endfor
24      return(CSupport, LSupport, Pruned , Dom)
endfunction
function AC-4 (CSupport, LSupport, Pruned, Dom)
1       stop:= empty_domain(Dom)
2       while (Pruned ≠ ∅and not stop) do
3               (X_i,a):= select_element(Pruned)
4               Pruned := Pruned  - {(X_i,a)}
5               (CSupport, Pruned , Dom):=
6                               propagate_del (X_i, a, CSupport, LSupport, Pruned, Dom)
7               if(empty_domain(Dom)) stop:= true endif
8       endwhile
9       return(CSupport, Dom)
endfunction
function propagate_del (X_i, a, CSupport, LSupport, Pruned, Dom)
10      forall (X_j,b) ∈LSupport_ia do
11                      CSupport_jbi := CSupport_jbi -1
12                      if(CSupport_jbi = 0) then
13                              Pruned := Pruned  ∪ {(X_j,b)}
14                              D_j:= D_j-{b}
15                      endif
16      endfor
17      return(CSupport, Pruned , Dom)
endfunction
```

Figure 2.3: AC-4 [Mohr and Henderson, 86].

Theorem 2.1: [Freuder, 82]

Given an arbitrary CSP:

1. A depth-first search ordering is backtrack-free if the level of strong k-consistency in the problem is greater than the width of the corresponding ordered graph.
2. There exists a backtrack-free depth-first search ordering for the problem if the level of strong k-consistency in the problem is greater than the width of the constraint graph.

This theorem suggests that a CSP can be solved by enforcing the corresponding level of local consistency and proceeding to the backtrack-free depth-first search. However, it is of practical interest only when constraint graphs have the lowest width (*i.e.*: 1). Otherwise, achieving the level of local consistency required adds new constraints to the graph. This increases the graph width, so higher levels of local consistency are required, and so on. This process loops until the local consistency enforcement does not increase the graph width. If we restrict ourselves to graphs of width 1, this process does not take place because the degree of local consistency required (arc-consistency) does not add new constraints, so the graph width is not modified. This is formalized in the following observation,

Observation 2.2:

Given an arbitrary CSP, if its associated graph has width 1 (*i.e.*: the graph is a tree) and it is arc-consistent, then there is a backtrack-free search.

Proof:

Since the graph is a tree, order the problem variables in such a way that for each variable, its parent is always before in the ordering. Observe that, under this ordering, the graph has width 1. Assign the first variable with an arbitrary value. Consider variables subject to the ordering and assign to each one a consistent value with respect previous variables. This consistent value always exists because each variable is only constrained with one past variable (width is 1) and the problem is arc-consistent. Thus, a consistent total assignment is found without any backtracking.

The first observation shows that tree-like CSP can be solved quite efficiently. Enforcing arc-consistency in a tree-like CSP using AC-4 (Figure 2.3) has a cost in $O(n \cdot m^2)$ because in trees $e = n\text{-}1$. The backtracking-free instantiation takes $O(n \cdot m)$ steps. Therefore, the whole process can be done in $O(n \cdot m^2)$.

Dechter and Pearl [Dechter and Pearl, 1988] observe that arc-consistency is stronger than necessary for enabling backtrack-free search

in tree-like CSP. They propose the concept of *directional arc-consistency*, which is a sufficient condition for backtrack-free search in trees (observe that directional arc-consistency is sufficient for the proof of Observation 2.2.1). Directional arc-consistency is defined upon a total ordering among variables.

Definition 2.3:

A CSP is *directional arc-consistent* under an ordering of the variables if and only if for every assignment $X_i \leftarrow v^i$ there exists a consistent assignment for every variable X_j which is after X_i in the ordering.

Figure 2.4 gives an algorithm for achieving directional arc-consistency. It is obvious that directional arc-consistency is a weaker condition than full arc-consistency. Thus, it provides a more efficient method for solving tree-like CSP.

Example 2.3:

Consider a CSP having three variables (*i.e:* $\{X_1, X_2, X_3\}$), three values per variable (*i.e:* $\{a, b, c\}$), and the two following constraints: $R_{12}=\{(b,b), (c,b)\}$, $R_{13}=\{(a,a), (c,a)\}$. It is easy to see that its constraint graph is a tree and, under lexicographical order, it has width 1.

Enforcing arc-consistency (using, for instance, AC-4 of Figure 2.3), some values are removed and the following domains are obtained: $D_1=\{c\}$, $D_2=\{b\}$, $D_3=\{a\}$. With these domains a backtrack-free search is guaranteed because the assignment of the first variable with its only value c, is necessarily consistent with the remaining values of the two other variables.

Enforcing directional arc-consistency (using *directional-arc-consistency* of Figure 2.4), fewer values are removed and the following domains are obtained: $D_1=\{c\}$, $D_2=\{a,b,c\}$, $D_3=\{a,b,c\}$. Assigning value c to the first variable, we still know that posterior variables have one consistent value with it, so we can deepen in the tree knowing that we will not need to undo the assignment.

A completely different line of work searching for tractable classes of CSP restricts the type of allowed constraints. In the following we give a brief review of some classes of constraints that have been found useful to characterize tractable CSP. Consider the following definitions,

Definition 2.4:

A constraint R_{ij} is *functional* if and only if for all $a \in D_i$ (respectively $b \in D_j$) there exists at most one $b \in D_j$ (respectively $a \in D_i$) such that $(a,b) \in R_{ij}$.

```
Function Directional-arc-consistency(Dom)
1       for i:=n-1 to 1 do
2               for j:=n to i+1 do
3                       for all a∈D_i do
4                               if(arc_inconsistent(X_i,a,X_j)) then D_i :=D_i -{a} endif
5                       endfor
6               endfor
7       endfor
8       return(Dom)
endfunction
function arc_inconsistent(X_i,a,X_j) returns boolean
9       arc_incons:=true
10      for all b ∈D_j while (arc_incons) do
11              if(not inconsistent(X_i←a, X_j←b)) then arc_incons:=false endif
12      endfor
13      return(arc_incons)
endfunction
```

Figure 2.4. Algorithm for the achievement of directional arc-consistency. It assumes lexicographical variable ordering [Dechter and Pearl, 88].

Definition 2.5:

A constraint R_{ij} is *monotonic* if and only if there exists a total ordering on D_i and D_j such that,

$$(a,b)\in R_{ij} \Rightarrow ((c,d)\in R_{ij} \ \forall c,d \text{ such that } c \leq a \text{ and } d \geq b)$$

Example 2.4:

The interest of functional and monotonic constraints relies on the fact that basic arithmetic constraints on the set of natural numbers are of one of these types. It is easy to see that the following equations belong to this class:

$$aX = bY + c$$
$$aX \leq bY + c$$
$$aX \geq bY + c$$

where upper case letters represent variables and lower case letters represent constants.

A generalization of functional and monotone constraints are row convex constraints. A constraint can be represented as a (0,1)-matrix defined upon a total ordering among values. The (a,b) entry on the matrix takes value 1 if the pair of values is permitted by the constraint (note that, in order to distinguish matrix rows from columns, we assume an order between variables i and j).

Definition 2.6:

A constraint R_{ij} is *row-convex* under a total ordering among values if and only if at each column of the corresponding matrix all 1's are consecutive.

Example 2.5:

Consider a simple constraint $R_{12} = \{(a,c), (b,a), (b,b), (b,c), (c,a)\}$. This constraint is row convex under lexicographical order because its associated matrix does not have any row with non consecutive 1's,

X_2 \ X_1	a	b	c
a	0	1	1
b	0	1	0
c	1	1	0

Finally, another kind of constraints found useful to characterize tractable CSP is the class of 0/1/all constraints defined as follows,

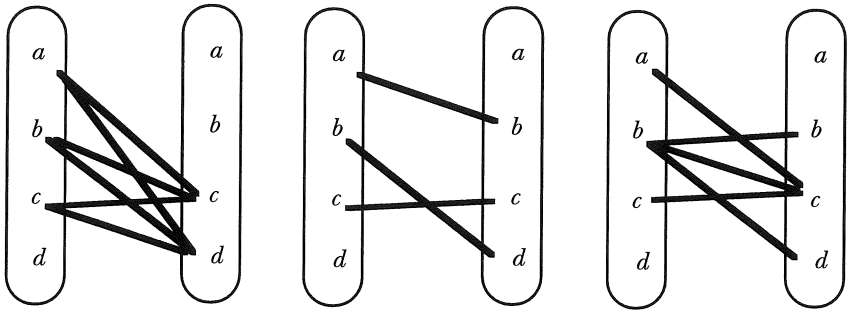
Definition 2.7:

A constraint is called *0/1/all* if and only if it belongs to any of the following types:

1. A *complete* constraint is a constraint $R_{ij} = \{A \times B\}$ for some $A \subseteq D_i$ and $B \subseteq D_j$.
2. A *permutation* constraint is a constraint $R_{ij} = \{(a, \pi(a): a \in A\}$ for some $A \subseteq D_i$ and some bijection $\pi: A \rightarrow B$, where $B \subseteq D_j$.
3. A *two-fan* constraint is a constraint R_{ij} where there exists $a \in A$ and $b \in B$ with $R_{ij} = \{(a \times B) \cup (b \times A)\}$, $A \subseteq D_i$ and $B \subseteq D_j$.

Example 2.6:

The following picture shows a typical complete constraint, permutation constraint and two-fan constraint. Solid lines connecting values indicate permitted values.



All these types of constraints have been found useful to characterize tractable classes of constraints. For instance, [Deville and Van Hentenryck, 91] show that if a CSP is limited to functional and monotonic constraints, arc-consistency is sufficient to test the problem satisfacibility. The interest of row-convexity is motivated in [van Beek, 91], where he shows that a consistent instantiation can be found in a backtrack-free manner in a path-consistent CSP such that there exists an ordering of the variables and the domains for which constraints are row-convex. [Cohen et *al.*, 94] show that if a CSP is limited to *0/1/all* constraints, enforcing arc-consistency produces a problem that can be solved without backtracking. In [Jeavons et *al.*, 95], it is shown that all known classes of constraints which lead to tractable problems can be characterized by a simple algebraic closure condition.

The introduction of functional and monotonic constraints has a practical impact in the complexity of the procedures achieving arc-consistency. There is a generic arc-consistency algorithm AC-5 [Deville and Van Hentenryck, 91], which can be instantiated to reduce AC-3 or AC-4 algorithms. In addition, when the problem only contains functional or monotonic constraints, AC-5 can be instantiated to produce an algorithm of complexity $O(e{\cdot}m)$. This is a significant improvement over the optimal algorithm in the general case which has a complexity of $O(e{\cdot}m^2)$.

2.2.3 Look-ahead Algorithms

Search algorithms can be combined with consistency enforcement algorithms looking for a better dead-end detection, or equivalently, detecting dead-ends at earlier levels in the tree. The idea is to enforce local consistency at each node during search. If the current node is in a dead-end and this is not detected by the search algorithm, achieving some level of local consistency may lead to its discovery. In this way, search does not need to unsuccessfully visit deeper nodes of the current subtree. The

process of enforcing local consistency is generally called the *look-ahead* or the *propagation* of the current assignment.

In practice, algorithms that perform a limited amount of propagation (enforce a low level of local consistency) are among the most effective. *Forward checking* (FC) [Haralick and Elliot, 80] is a simple, yet powerful algorithm for total constraint satisfaction. It propagates the effect of each assignment by pruning from domains of future variables those values that are inconsistent with the assignment.

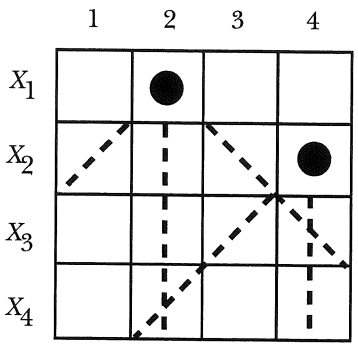
Figure 2.5 shows FC. It works like BT, except that future domains dynamically change. At a given node, only those values consistent with past assignments remain feasible. Consequently, after each new assignment, future domains are pruned. When a future domain becomes empty, FC backtracks because there is no value for one future variable consistent with the current assignment. The function in charge of value pruning is *look_ahead*. It iterates on future domains removing unfeasible values and returns updated domains.

```
function FC (P, F, Assg, Dom) returns boolean
1       if (F=∅) then
2               Sol := Assg
3               return (true)
4       endif
5       (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6       stop:= false
7       while (D_i≠∅ and not stop) do
8               a:= select_current_value(D_i)
9               D_i:= D_i-{a}
10              NAssg:= Assg ∪ {X_i←a}
11              NDom:= look_ahead(X_i, a, F-{X_i}, Dom-{D_i})
12              if (not empty_domain(NDom)) then
13                      stop:=FC(P∪{X_i}, F-{X_i}, NAssg, NDom)
14              endif
15      endwhile
16      return(stop)
endfunction
function look_ahead(X_i, a, F, Dom)
17      stop:= false
18      for all D_j ∈Dom while (not stop) do
19              for all b ∈D_j do
20                      if (inconsistent(X_i←a, X_j←b) then D_j := D_j -{b} endif
21              endfor
22              if(D_j  = ∅) then stop:= true endif
23      endfor
24      return (Dom)
endfunction
```

Figure 2.5: Forward Checking [Freuder and Wallace, 92].

Example 2.7:

Consider the 4-queens problem and a search state with the following partial assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 4\}$. At this point, FC has the following sets of future values: $D_3 = \{1\}$ and $D_4 = \{1,3\}$. This situation is depicted in the following board configuration,



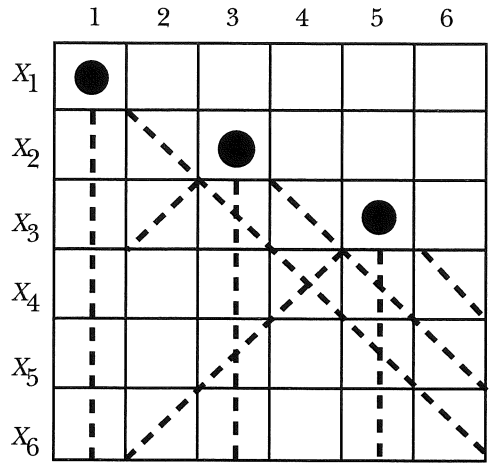
Since there is no empty domain, search continues with a new assignment. If X_3 is the new current variable, it only has one feasible value so the new search state is defined by $\{X_1 \leftarrow 2, X_2 \leftarrow 4, X_3 \leftarrow 1\}$. The look-ahead of the current assignment removes value 1 from D_4, but does not cause any empty domain because $3 \in D_4$ remains feasible. Since it is the only value left for X_4, it is assigned. The new assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 4, X_3 \leftarrow 1, X_4 \leftarrow 3\}$ is a problem solution.

For many years it was believed that performing higher levels of propagation than forward checking was not cost effective [Kumar, 92]. However, recent research on total constraint satisfaction has contradicted this belief. In [Sabin and Freuder, 94; Bessière and Régin, 96] there is strong experimental evidence of the important savings that maintaining full arc-consistency during search can produce.

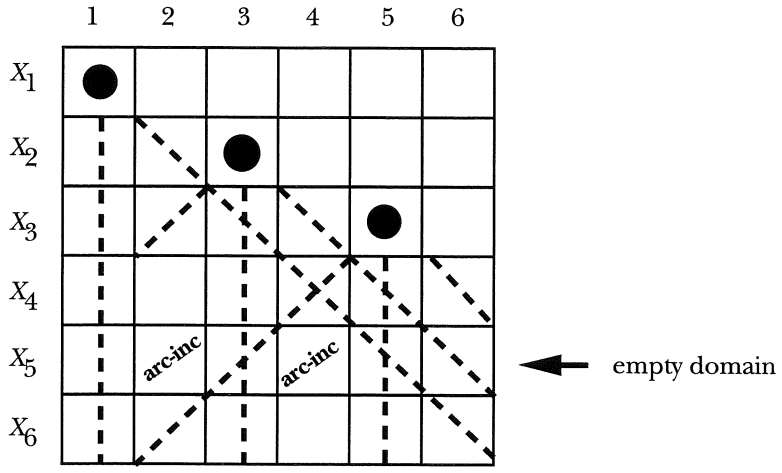
The algorithm that maintains arc-consistency during search, denoted MAC [Sabin and Freuder, 94], requires more computational effort than FC at each search state. This additional computation has two objectives: (*i*) arc-inconsistent values are filtered, so the subproblem search space is simplified; (*ii*) if the propagation process causes an empty domain, then the subproblem is unsolvable. Given that MAC can prune more values than FC, it has better dead-end detection capabilities. This causes that MAC can backtrack in nodes where FC would continue searching below. In easy problems, MAC is not the most efficient algorithm because the tree reduction does not pay off the effort of maintaining arc-consistency. However, on hard problems MAC is considered the most efficient algorithm. In this case, maintaining arc-consistency is cost effective.

Example 2.8:

Consider the 6-queens problem and a search state defined by a partial assignment $\{X_1 \leftarrow 1, X_2 \leftarrow 3, X_3 \leftarrow 5\}$. Forward checking would be in the following situation,



At this point FC would continue search because no dead-end is detected. In fact, assuming lexicographical variable and value ordering, FC would need to deepen two more levels in the tree before detecting the dead-end. On the other hand, if we enforce arc-inconsistency in the current subproblem, we would have the following situation,



An arc-consistency algorithm would detect that value b of X_5 does not have any consistent value in X_4, so it can be removed. Besides, value

4 of X_5 does not have any consistent value in X_6, so it can also be removed. Consequently, no value remains feasible for X_4, so the subproblem is arc-inconsistent (and, therefore, unsolvable). Hence, the algorithm can safely backtrack at this point without further search.

MAC can be implemented using any arc-consistency algorithm. In Figure 2.6, we present a simple implementation based on AC-4 (namely, MAC-4). It uses the AC-4 algorithm described in Figure 2.3. MAC-4 receives as parameters the sets of past and future variables, the current assignment, and the AC-4 data structures (*i.e: CSupport* and *LSupport*). It assumes that the problem is initially arc-consistent, and all support-based data structures are properly initialized. It works like forward checking except in two points:

1. The look-ahead returns not only the new domains, but the set of pruned values (line 11). This set is used as the input of an AC-4 call, which propagates their deletion and enforces arc-consistency at the current subproblem (line 13).
2. After unsuccessfully attempting an assignment, it has been shown to be unfeasible subject to the set of past variables. MAC-4 removes that value and propagates its deletion, enforcing arc-consistency (line 18).

2.2.4 Look-back Algorithms

There are a number of ways in which the basic BT strategy can be improved. For simplicity, we assume in the following that depth-first selects variables and values in lexicographical order, and consistency of the current assignment is also checked with past variables in lexicographical order.

Backmarking (BM) [Gaschnig, 77] avoids the repetition of some consistency checks. When BT assigns the current variable $X_i \leftarrow a$ it checks the consistency of this assignment with past variables $\{X_1 \leftarrow v^1, \ldots, X_{i-1} \leftarrow v^{i-1}\}$. If any of the tests fails, BM records the point of the failure in an array mcl_{ia} (*maximum check level*). Consider that the algorithm backtracks up to variable X_k, deepens in the tree and attempts again the same assignment $X_i \leftarrow a$. In this situation it is known that the current assignment is consistent with past variables up to mcl_{ia} as far as their assignment has not been changed. BM avoids the repetition of these already performed checks. To do this effectively, BM needs an additional array mbl_i (*minimum backtrack level*) which for each variable records how far it has backtracked since the last attempt to instantiate this variable.

Backjumping (BJ) [Gaschnig, 78], improves BT by making a more suitable decision of what variable backtrack to. BJ only differs from BT at those nodes where a dead-end is detected. Instead of backtracking to the most recently instantiated variable, BJ *jumps* to the deepest past variable

```
function MAC-4 (P, F, Assg, Dom, CSupport, LSupport) returns boolean
1     if (F=∅) then
2           Sol := Assg
3           return (true)
4     endif
5     (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6     stop:= false
7     while (D_i≠∅ and not stop) do
8           a:= select_current_value(D_i)
9           D_i:= D_i-{a}
10          NAssg:= Assg ∪ {X_i←a}
11          (NDom, Pruned) := look_ahead(X_i, a, F-{X_i}, Dom-{D_i})
12          if (not empty_domain(NDom)) then
13                (NewCSupport, NDom) := AC-4(CSupport, LSupport, Pruned, NDom)
14                      if (not empty_domain(NDom)) then
15                       stop:=MAC-4(P∪{X_i}, F-{X_i}, NAssg,NDom, NewCSupport,
                                    LSupport)
16                      endif
17          endif
18          if(not stop and D_i ≠∅) then
                  (CSupport, Dom) := AC-4(CSupport, LSupport, {(i,a)}, Dom) endif
19    endwhile
20    return(stop)
endfunction
function look_ahead(X_i, a, F, Dom)
21    Pruned:= ∅
22    stop:= false
23    for all D_j ∈Dom while (not stop) do
24          for all b ∈D_j do
25                if (inconsistent(X_i←a, X_j←b) then
26                      D_j := D_j -{b}
27                      Pruned := Pruned ∪ (j,b)
28                endif
29          endfor
30          if(D_j  = ∅) then stop:= true endif
31    endfor
32    return (Dom, Pruned)
endfunction
```

Figure 2.6: Basic version of MAC-4.

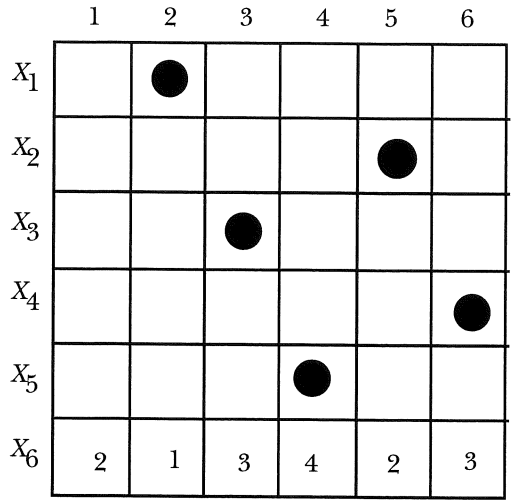
that the current variable was checked against. The reason is that changing the assignment made to variables in between does not change the culprit for the dead-end, so there is no point in visiting those nodes. When the current variable is not responsible for the dead-end detection, no jump is done. In that case BJ backtracks chronologically.

Conflict-Directed Backjumping (CBJ) [Prosser, 93a] improves BJ by following a more sophisticated jumping strategy. The conflict set of a variable is formed by the past variables with which a consistency check failed with some value of the considered variable. For instance, if the current assignment $X_i \leftarrow a$ is inconsistent with the past assignment $X_k \leftarrow v^k$, variable X_k is added to the conflict set of X_i. When all values have

been attempted for the current variable, CBJ jumps to the deepest variable in its conflict set. This variable is removed from the conflict set of the current variable, and this new conflict set is added to the conflict set of the variable it jumps to. With this approach, jumps can be done at those nodes where backtracking occurs not because a dead-end is detected, but because all values have already been attempted. In addition, more that one jump can be done along the same path from a detected dead-end to the root.

Example 2.9: [Kondrak and van Beek, 97]

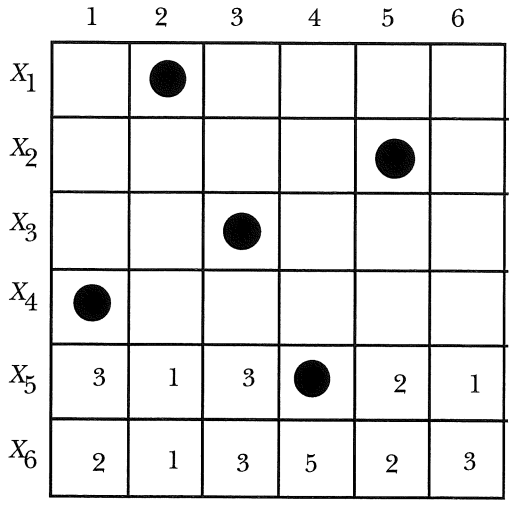
Consider the 6-queens problem and a node defined by the assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 5, X_3 \leftarrow 3, X_4 \leftarrow 6, X_5 \leftarrow 4\}$. At this point, X_6 is selected for instantiation, and a dead-end is detected because all its values are found inconsistent with some past variable. BJ records the set of variables with which some value of X_6 is found inconsistent. The following picture illustrates the situation and indicates for each value of X_6 the inconsistent past variable.



In this situation, BT would backtrack to X_5 and would attempt the assignment $X_5 \leftarrow 5$. However, BJ detects that the assignment made to X_5 is not causing the dead-end and jumps to X_4, which is the deepest variable contributing to the detected dead-end. Since no more values for X_4 are left, BJ backtracks to X_3 and attempts the assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 5, X_3 \leftarrow 4\}$.

Consider now the node defined by the assignment $\{X_1 \leftarrow 2, X_2 \leftarrow 5, X_3 \leftarrow 3, X_4 \leftarrow 1, X_5 \leftarrow 4\}$. At this point, CBJ records the conflict set of X_5 which is formed by the set of past variables inconsistent with its previously attempted values. It is easy to see that this conflict set is $\{X_1, X_3\}$. In this situation, X_6 is selected for instantiation and a

dead-end is detected because all its values are found inconsistent with some past variable. The conflict set computed for X_6 is $\{X_1, X_2, X_3, X_5\}$. At this point, BT, BJ and CBJ backtrack to X_5. However, CBJ updates the conflict set of X_5 producing the new set $\{X_1, X_2, X_3\}$. After unsuccessfully attempting the two remaining values of X_5, both BT and BJ backtrack chronologically to X_4. But CBJ jumps to X_3 because it is the deepest variable in its conflict set. The following picture illustrates the situation and indicates for each inconsistent value of X_5 and X_6 the past variable against which inconsistency is detected.



2.2.5 Heuristics

Depth-first algorithms for partial and total constraint satisfaction do not specify the order in which variables and values are selected. It is well known that these orderings have a dramatic effect on the algorithms' efficiency [Dechter and Meiri, 94]. Thus, before using any constraint satisfaction algorithm, a variable and a value ordering heuristic must be chosen. Heuristics can be grouped into two categories:

- *Static orderings:* A static heuristic establishes an ordering before search starts, and maintains this ordering throughout all of the search. Thus, using static variable and value orderings the search tree has a fixed structure. Following any path, one finds the same variables at the same tree levels; traversing any tree level, one finds the same sequence of values repeated at each subtree.
- *Dynamic orderings:* A dynamic heuristic makes selections dynamically during search. Thus, using dynamic variable and value orderings the next current variable and the order in which its values are assigned is

decided at each search node. Using dynamic orderings, search trees are more flexible because one finds different variables assigned at the same tree level. In addition, values may be attempted in a different order each time a variable is selected.

A well-known static heuristic involves ordering variables by their constraint graph degree. The idea is to consider first the most constrained variables because they are likely to be more difficult to assign. Inconsistencies are expected to be found at early tree levels, where recovering from mistakes is less costly. Variables with few constraints have more freedom in the values they can take, so it is easier to find a good value for them. With this heuristic their assignment is delayed to deep tree levels. This static variable ordering, denoted *maximum degree ordering heuristic* , has been often used in the literature.

The *minimal bandwidth ordering heuristic* [Zabih, 90] gives a static order to variables under which the graph bandwidth is low (the bandwidth of a node under an ordering is the maximum distance between the node and any other node adjacent to it; the bandwidth of a graph is the maximum bandwidth among its nodes). The intuition behind this heuristic is that constrained pairs of variables are likely to be close. Thus, if the assignment of one causes a failure in the other, the backtracking will not be very costly. The main disadvantage of this heuristic is that finding variable orderings with low bandwidth is computationally expensive.

It is strongly believed in the CSP community that dynamic variable orderings are more effective than static ones. The most popular variable ordering heuristic selects the variable with the minimum number of values in its current domain [Haralick and Elliot, 80]. This heuristic, denoted *minimum domain* (MD), is usually applied to look-ahead algorithms because the actual size of domains is given at no cost. Using probabilistic methods, Haralick and Elliot show that MD minimizes the expected tree size. However, it is important to point out that they assume a uniform probability of finding a good value for every variable. Moreover, the probability of choosing a good value for a variable is independent of previous assignments to past variables.

The performance of MD is often improved with the addition of some information from the graph topology. For instance, [Frost and Dechter, 95] use *graph degree* to discriminate among variables with the same domain size; [Bessière and Régin, 96] select the variable having the lowest ratio *domain cardinality divided by degree* in an attempt to combine both dynamic and static information.

Value ordering heuristics have not attracted the attention of the CSP community as much. It is generally believed that good values are those which are more likely to participate in solutions. This idea is developed in [Dechter and Pearl, 88] where they propose a value ordering heuristic which relies on a tree-relaxation of the problem to estimate the goodness of a value. At a given node, the minimum spanning tree of the subgraph restricted to future variables is computed. This tree-like graph is used as a

relaxation of the original problem for which solutions can be found efficiently (as indicated in Observation 2.2). Assuming that good values for the relaxed problem are also good values for the original problem, they sort values by the number of solutions in which they participate.

A different approach for value ordering is followed in [Keng and Yun, 89; Geelen, 92; Frost and Dechter, 95]. Within the context of look-ahead algorithm, values are ordered by the pruning effect that they have on future domains. This approach requires the propagation of each possible assignment to obtain the size of the resulting domains. For job-shop problems, [Keng and Yun, 89] sort values by increasing *cruciality* which is the sum over future variables of the ratio of values that the assignment prunes. Cruciality is justified as an impact measurement on the available resources. In [Geelen, 92] values are sorted by increasing *promise* which also uses the ratio of pruned values, but the contribution of each future variable is multiplied rather than added. Promise is justified as an upper bound of the number of possible solutions. In [Frost and Dechter, 95] the cardinality of the resulting domains is added and values are sorted by decreasing number of surviving values.

2.2.6 Combined approaches

Often times, CSP techniques do not have to be competitors, but rather can be combined in a cooperative way. A good example of how different algorithms can be combined is shown in [Prosser, 93a]. Starting from four classical algorithms: BJ, CBJ, BM and FC, he develops four hybrid algorithms: BM-BJ, BM-CBJ, FC-BJ and FC-CBJ, which combine features and take advantage of the best of each one.

In the same way, local consistency algorithms can be combined to search both as a pre-processing and at each node. MAC is a good example that was already presented in Section 2.2.3. Another example is MAC-CBJ [Prosser, 95], which is a hybrid of MAC and CBJ.

The same idea can be extended to heuristics because they are usually justified in an algorithm independent way. Most variable and value ordering heuristics can be combined. For instance, MD which was presented in the context of FC, can be applied to any CSP algorithm [Bacchus and van Run, 95].

2.3 Algorithms for Partial Constraint Satisfaction

2.3.1 Depth-First Branch and Bound

Branch and Bound is an algorithmic schema for optimization problems that can be used to solve MAX-CSP. It uses the search tree defined in Section 2.1. Using branch and bound terminology, the goal is to find the tree leaf that minimizes a cost function. In the MAX-CSP case, the cost function is the number of violated constraints. In the search tree, each node has associated with it a (possibly inconsistent) partial assignment. The number of constraints violated by the node assignment is called its *distance*. Branch and bound do a depth-first tree traversal. During the traversal, the algorithm keeps track of the best solution found so far which is the total assignment with minimum distance in the explored part of the search tree. Its distance is used as an *upper bound* of the allowable cost. In addition, the algorithm uses the current node distance as an underestimation of the best solution that can be obtained searching below. This value is a *lower bound* of what can be achieved following the current line of search. Branch and bound is in *a dead-end* when it visits a node which does not have any leaf below that improves the current upper bound. The dead-end is detected when the current *lower bound* is greater than or equal to the current *upper bound* because at this point we know that the current path cannot lead to a better solution than the current best one. In this case, branch and bound backtracks to a previous node.

Figure 2.7 shows *partial chronological backtracking* (PBT), the partial consistency counterpart of BT. P and F denote past and future variables. *dist* is the current assignment distance. *Assg* is the current assignment. Dom are future variable domains. From an initial call $PBT(\{X_1,...,X_n\},\varnothing,0,\varnothing,\{D_1,...,D_n\})$, the algorithm searches for the best problem solution, which is reported by means of the global variable *Best_sol*. Its distance is kept in another global variable, *UB*. If the set of future variables is empty, the current assignment is the best solution found up to this point, so it is recorded and the upper bound is updated (lines 2 and 3). If there are future variables, one of them is selected (line 5), and its values are sequentially attempted. For each assignment, the inconsistencies that it has with past assignments are computed and added to the current distance (line 10). If the current distance is lower than the current upper bound a recursive call is made (line 12). Otherwise, the next value is attempted.

```
procedure PBT (P, F, dist, Assg, Dom)
1       if (F = ∅) then
2               UB:= dist
3               Best_sol:= Assg
4       else
5               (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6               while (D_i ≠ ∅) do
7                       a:= select_current_value(D_i)
8                       D_i:= D_i-{a}
9                       NAssg:= Assg ∪ {X_i←a}
10                      ndist:= dist + cost( X_i, a, Assg)
11                      if (ndist < UB) then
12                                              PFC( P∪{X_i}, F-{X_i}, ndist, NAssg, Dom-{D_i})
13                      endif
14              endwhile
15      endif
endprocedure
function cost( X_i, a, Assg) returns boolean
16      cost:=0
17      for all (X_j←v^j)∈Assg do
18                      if (inconsistent(X_i←a, X_j←v^j))) then cost:=cost+1 endif
19      endfor
20      return (cost)
endfunction
```

Figure 2.7: Partial Chronological Backtracking [Freuder and Wallace, 92].

2.3.2 Look-ahead

The forward checking counterpart for MAX-CSP, denoted *partial forward checking* (PFC) [Freuder and Wallace, 92], uses forward checking to improve the lower bound and detect unfeasible future values. PFC look-ahead keeps for all feasible values of future variables the number of inconsistencies with previous assignments. The *inconsistency count* (IC) associated with value b of variable X_j, ic_{jb}, is the number of inconsistencies that value b has with the current assignment of past variables. The sum of minimum IC, $\sum_{j \in \mathbf{F}} min_v\{ic_{jv}\}$, is a lower bound of the number of inconsistencies that will necessarily occur in constraints between past and future variables if the current partial assignment is extended to a total one (no matter what values are assigned to future variables). Therefore, it can be added to the current distance in order to obtain a better lower bound at the current node. Thus,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} \qquad (2.1)$$

is the lower bound computed by PFC, where *distance* is the number of inconsistencies among past variables.

Each assignment is propagated toward future ICs. During the propagation, ICs are also used to detect unfeasible values. Thus, a future value b of variable X_j, has the following associated lower bound,

$$distance + ic_{jb} \tag{2.2}$$

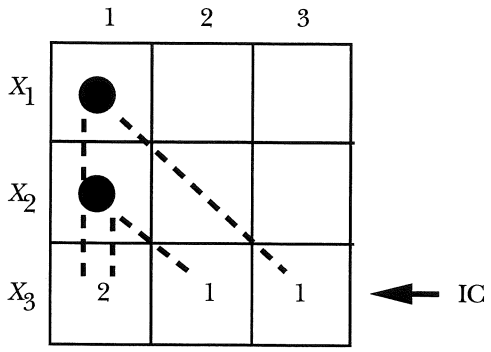
If the lower bound of a future value is greater than or equal to the current upper bound, it can be pruned because its assignment cannot produce any solution improving the current upper bound. If the look-ahead produces an empty domain, the algorithm backtracks. Expression (2.2) was the pruning condition for PFC as described in [Freuder and Wallace, 92]. However, one should note that a stronger pruning power is obtained if the following lower bound is used

$$distance + ic_{jb} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} \tag{2.3}$$

As far as we know, no one has previously reported this *obvious improvement* on PFC. Nevertheless, it may be the case that some researchers already include it in their implementations, although still referring to PFC in the description of their experiments. In this work we take it for granted.

Example 2.10:

Consider the 3-queens problem. It was shown in Example 2.2 that it is an overconstrained problem. A search state characterized by the following assignment $\{X_1 \leftarrow 1, X_2 \leftarrow 1\}$ produces the following situation,



where we indicate at future value cells their inconsistency count (IC) subject to the current assignment. Thus, $ic_{31} = 2$ because it is inconsistent with the assignment made to two past variables. The current assignment has distance 1 because there is a constraint

violation between the two past variables. At this point, PFC lower bound is,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} = 1 + 1 = 2$$

which means that any extension of the current partial assignment to a total one, will violate at least two constraints. If the algorithm has previously found a solution with distance lower than or equal to 2, it can backtrack at this point.

Figure 2.8 presents PFC. It requires a new parameter *IC*, which keeps the current inconsistency counts. At each node, PFC selects one variable and attempts the consecutive assignment of its feasible values. The new distance is computed by adding the corresponding IC to the current distance (line 10). Right after attempting a new assignment, the feasibility of the current value is tested (line 11). It is the last chance for pruning a value before propagating its assignment. This final feasibility test is required because there may have been an upper bound decrease at a sibling subproblem. If the current assignment is feasible, it is propagated to all feasible future values updating their IC and pruning them if they are detected to be unfeasible. Procedure *look_ahead* is in charge of the propagation. It returns the new domains and IC.

From the algorithm, it may seem as if PFC has two different dead-end conditions: when the lower bound becomes greater than or equal to the upper bound (line 11) and when the look-ahead produces an empty domain (line 13). But it is not so because both conditions are basically the same thing, as shown in the following observation.

Observation 2.3:

Consider that look-ahead has produced an empty domain in a future variable X_j. Therefore, the following condition holds for all its values,

$$distance + ic_{jb} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} \geq upper\_bound \qquad \forall b \in D_j$$

particularly, it also holds for the value having the minimum IC,

$$distance + min_v\{ic_{jv}\} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} \geq upper\_bound$$

Thus, when the look-ahead produces an empty domain, it means that with the new ICs the lower bound has become greater than or equal to the upper bound.

```
procedure PFC ( P, F, dist, Assg, Dom, IC)
1       if (F = ∅) then
2               UB:= dist
3               best_sol:= Assg
4       else
5               (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6               while (D_i≠∅) do
7                       a:= select_current_value(D_i)
8                       D_i:= D_i-{a}
9                       NAssg:= Assg ∪ {X_i←a}
10                      ndist:= dist + ic_ia
11                      if (ndist + Σ_{j∈F-{i}} min_v{ic_jv} < UB) then
12                              (NDom,NIC):= look_ahead(ndist, X_i, a, F-{X_i}, Dom-{D_i}, IC)
13                              if (not empty_domain(NDom)) then
14                                      PFC(ndist, P∪{X_i}, F-{X_i},ndist, NAssg, NDom,
NIC)
15                              endif
16                      endif
17              endwhile
18      endif
endprocedure
function look_ahead(ndist, X_i, a, F, Dom, IC)
19      stop:= false
20      for all D_j∈Dom while (not stop) do
21              for all b ∈D_j do
22                  if (ndist + ic_jb + Σ_{k∈F-{j}} min_v{ic_kv} ≥ UB) then
23                          D_j:= D_j - {b}
24                  else_if (inconsistent(X_i←a, X_j←b)) then
25                          ic_jb:=ic_jb+1
26                          if (ndist + ic_jb + Σ_{k∈F-{j}} min_v{ic_kv} ≥ UB) then
27                                  D_j:= D_j-{b}
28                          endif
29                  endif
30              endfor
31              if (D_j= ∅) then stop:= true endif
32      endfor
33      return (Dom, IC)
endfunction
```

Figure 2.8: Partial Forward Checking based on [Freuder and Wallace, 92].

2.3.3 Improvements to Look-ahead

Branch and bound effectiveness is strongly related to the quality of its bounds. One way to enhance PFC is to improve its lower bound adding information about new detectable inconsistencies. In this way, dead-ends can be detected at higher tree levels and search does not need to unsuccessfully visit deeper nodes of the current subtree. A method for computing higher lower bounds that makes use of local consistency information was presented in [Wallace, 94]. This approach uses the concept of *directed arc-consistency* (Definition 2.3). Given an arbitrary but fixed variable ordering, the *directional arc-inconsistency count* (DAC) of a value b of a variable X_j, dac_{jb}, is the number of variables which are arc-inconsistent with value b for X_j and appear after X_j in the ordering. Note that dac_{jb} is a lower bound of the number of inconsistencies that X_j will have if b is assigned to X_j.

In Wallace's approach, DAC counts are computed in a pre-processign step and added to the current node lower bound in the following way,

$$distance + \sum_{j \in \mathbf{F}} min_b\{ic_{jb}\} + \sum_{j \in \mathbf{F}} min_b\{dac_{jb}\}$$

The use of this lower bound requires that branch and bound instantiates its variables in the same fixed order that was used for DAC computation.

Recently, a new way of improving the lower bound using information from constraints relating future variables has been presented [Verfaillie et al., 96]. Their algorithm, denoted *russian doll search* (RDS), consists in solving an increasingly larger sequence of subproblems such that each subproblem is included in the subsequent ones, so information of smaller subproblems can be exploited. The sequence of subproblems starts from the initial problem restricted to two variables and ends with the initial problem with all its variables. At each step, one more variable is added. When solving a subproblem, the cost of the best solution of the previous smaller subproblem is used to improve the lower bound in the following way,

$$distance + \sum_{j \in \mathbf{F}} min_b\{ic_{jb}\} + rds^{\mathbf{F}}$$

where $rds^{\mathbf{F}}$ is the distance of the best solution of the problem defined by the set of future variables and the constraints among them. Again, this method requires the use of a static variable ordering.

2.3.4 Heuristics.

In the partial constraint satisfaction context, not much work about heuristics has been done. Regarding variable orderings, [Freuder and Wallace, 92] propose a dynamic variable ordering heuristic for PFC in which the variable having the highest mean of inconsistency counts among its feasible values is selected first. In [Wallace and Freuder, 93] the concept of graph width (Definition 2.2) is used. They propose the use of a *maximum node width* heuristic, which sequentially select the variable with the greatest number of constraints in common with variables already chosen. With this idea, a static ordering among variables is generated. This ordering is used as a primary criterion which is enhanced with a second criterion to break ties. The following tie breakers are proposed: minimum domains, maximum degree and largest mean of arc-inconsistency counters. A static variable ordering that has been found effective for random problems is to select variables by forward degree breaking ties with backward degree [Larrosa and Meseguer, 96]. At a search state, the forward degree of a future variable is the number of other future variables with which it is constrained. The backward degree of a future variable is the number of past variables with which it is constrained. It should be noted that in MAX-CSP, the dominance between dynamic and static variable heuristics is still a subject of research[2].

Regarding value ordering heuristics, [Freuder and Wallace, 92] propose to select values by increasing number of inconsistency counts (using PFC). In [Wallace and Freuder, 93; Wallace, 94], values are selected by increasing number of arc-inconsistencies. Arc-inconsistencies are computed prior to the search.

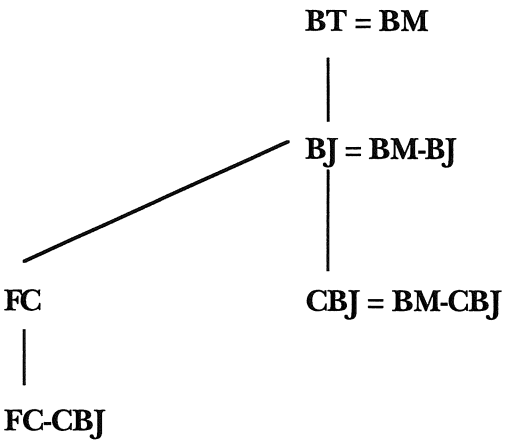


Figure 2.9: The hierarchy with respect to the number of visited nodes.

[2]In this Theses we will present some examples of this affirmation.

2.4 Algorithms Evaluation

2.4.1 Theoretical Evaluation

Historically, CSP algorithms have been compared and assessed on an experimental basis. The limitations of experimental evaluation have often been recognized [Prosser, 93a]. Experimental evaluation assumes that algorithmic performance on a reduced number of problem instances will extrapolate to larger and more practical classes of problems. It is clear that this is a strong assumption. However, theoretical evaluations have been found difficult to perform.

In that context, the recent work of [Kondrak and van Beek, 97] is especially relevant. They show that some classical algorithms can be ranked according to their efficiency using purely theoretical results. Basically, they characterize the behaviour of some CSP algorithms by the formulation of necessary and sufficient conditions for visiting a node. This characterization enables them to construct a hierarchy of the algorithms with respect to two standard performance measures: the number of visited nodes, and the number of performed consistency checks. These hierarchies indicate that an algorithm will never perform worse than another with respect to the corresponding search effort parameter. Figure 2.9 and Figure 2.10 depict the hierarchies obtained with respect to the number of nodes and the number of checks, respectively. Besides the relationships that are shown explicitly, one should be aware of those that are implicit in the picture. For pairs of algorithms that are not linked, there is no dominance between them. For instance, BM performs fewer consistency checks than FC on the 8-queens problem, but more on a variation of the problem called the confused 8-queens.

Theoretical comparison of algorithms has its own limitations. For many pairs of algorithms, one may find examples in which one outperforms the other, and the way around. In that situation, a theoretical analysis like the one presented in [Kondrak and van Beek, 97] is not suitable. Moreover, the importance of showing that one algorithm is better than another depends very much on the gain ratio, and theoretical analysis do not usually give this type of information. Therefore, theoretical and experimental evaluation must be taken as complementary rather than alternative ways to assess algorithms.

Typically, when different algorithms need to be empirically compared, a representative benchmark is chosen and the competing algorithms are tested on it. Their performance is measured in terms of some search effort parameters. From these experiments, one usually detects classes of problems for which some algorithm outperforms another. The superiority of algorithms is therefore assessed in terms of the efficiency

gain on the class, and the class generality. For instance, forward checking is superior to chronological backtracking in most problem classes [Bacchus and van Run, 95], and MAC is superior to forward checking on hard random problems [Bessière and Régin, 96].

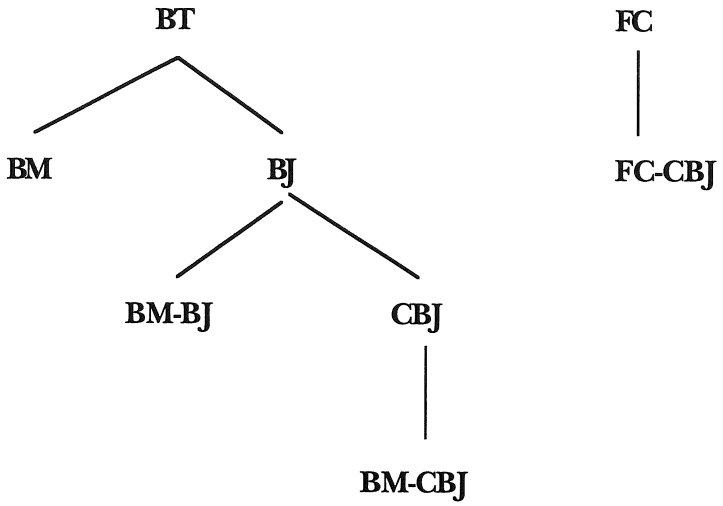


Figure 2.10: The hierarchy with respect to the number of consistency checks.

2.4.2 Random Problems

It is generally believed that efficiency can be gained by encoding domain specific knowledge into the problem solver. However, there are good reasons for studying general algorithms. First, specific algorithms are costly to develop and tune. Second, specific algorithms have limited application. Finally, general algorithms can be a start-point for the development of specialized technics.

Constraint satisfaction algorithms are usually conceived as general purpose techniques, applicable to a wide range of real problems. For this reason, there is a tradition of evaluating constraint satisfaction algorithms using random problems. They offer the following advantages:

- Random problems are easily generated, so experiments can be performed on large samples.
- It is easy to vary systematically some model parameters and observe the algorithm's behaviour subject to the change.
- Random problems do not have any domain-dependent bias. One then expects that conclusions drawn from the experiments will extrapolate to many different domains.
- Random problems have recently become even more popular after the discovery of the *phase-transition* phenomenon. The reason is

that now it is easy to characterize and generate average-case exponentially hard problem instances.

One may think of many different CSP random models. In this work we will use the well-known four parameter model because it has been used by most researchers in the CSP community for the last few years [Prosser, 94; Smith, 94]. In this model, random binary CSP are characterized by four parameters $<n, m, p_1, p_2>$, where n is the number of variables, m is the number of values for each variable, p_1 is the proportion of variable pairs which are constrained (graph *connectivity*), and p_2 is the proportion of forbidden value pairs between two constrained variables (constraint *tightness*). When constructing a random instance, we select $n(n\text{-}1)p_1/2$ of the possible variable pairs at random; for each selected pair we establish a constraint between the corresponding variables. For each constraint, we choose m^2p_2 value pairs at random as the forbidden values for that constraint. In the following, we use $<n, m, p_1, p_2>$ as the class of random instances generated in the form described above with fixed n, m, p_1 and p_2. If some parameter is missing it means that this parameter is varying freely. Using this model, if n, m and p_1 are kept fixed and p_2 is varied, there is a point at which problems suddenly change from being solvable to unsolvable. The hardest instances on average occur precisely at this point (this phenomenon is further discussed in Section 4.9.1).

It must be recognized that using random problems to evaluate algorithms has some disadvantages, too. We claimed that being domain independent was a point in favor of random problems. However, it may be argued the other way around. Their random nature makes random problems *different from any particular domain.* Thus, there is a risk of developing techniques that are specialized for random problems and do not work so well on particular domains. Specific domains have semantics behind their constraints which are usually reflected in an underlying structure at the corresponding CSP. In addition, within their random structure, random problems are artificially homogeneous. Every variable has the same number of values in its domain, every constraint has the same number of nogoods, every variable has the same expected degree, every value in a domain has the same expected constraining behaviour as the other values, etc. Such homogeneity does not occur in many real applications and one has to be aware that the algorithms' performance may change with different domains.

To conclude, we consider random problems as an appropriate model to empirically evaluate general purpose techniques and we will use it as our main benchmark throughout this work. However, we think that algorithms should not only be tested on random problems. For this reason, we usually make additional experiments on other domains.

2.4.3 Search Effort Measures

When comparing the performance of different search procedures, all of them having asymptotically equivalent computational requirements, it is clearly important to consider appropriate computational effort parameters. There are three measurements frequently used in the literature. However, all of them have their advantages and disadvantages:

- CPU time: The amount of CPU required during search is clearly relevant to computational effort. However, it is highly environment and implementation dependent and often times difficult to measure accurately. Furthermore, mainly due to the development of computers and networks, in most research laboratories it is difficult to reproduce the same environment for different executions.
- Visited nodes: Since we are concerned with tree search algorithms, we can evaluate them measuring the number of nodes that they visit. Each time the algorithm attempts to instantiate a variable, it is counted as a visited node. The advantage of this measure is that it is both environment and implementation independent. However, different algorithms may perform a non comparable amount of work at each node. For example, simple backtracking performs an $O(n)$ task at each node that it visits, forward checking complexity at each node is $O(nm)$, and MAC complexity per node can be up to $O(em^2)$. For this reason, the number of visited nodes is better taken as a measure of the algorithms' *search quality*. Obviously, there is a trade-off between the search quality and the cost of achieving it.
- Consistency checks: A consistency check takes place each time that the algorithm checks whether two values can be consistently assigned to two constrained variables. Most algorithms are defined in terms of how they maintain consistency at each node. For this reason, the cost of visiting each node can be established in terms of the number of consistency checks that it performs. Consistency checks are probably the best environment independent measure for computational effort. However, one should be cautious about taking consistency checks as the only measurement because different algorithms have different overhead apart from checking consistency. In addition, it should be mentioned that sophisticated algorithms use specialized data structures (such as support-lists in AC-4) to *keep* some consistency information and speed up its retrieval. Counting consistency checks may not be appropriate for these cases. Nevertheless, the number of consistency checks is the most frequently used measurement and we will use it as the principal search effort indicator.

2.5 Historical Account

The work presented in this dissertation includes results obtained during the period of time between 1995 and 1998. Because of the area dynamism, we find that the current state-of-the-art differs significantly from what was in 1995, when our work was started. For this reason, we believe that it is important to place our work in its appropriate historical context. In this Section we summarize what were the *hot* topics related to our work when it was started, and what progress has occurred parallel to our research.

2.5.1 Research Trends in 1995

A topic generating controversy in 1995 was the possible superiority of stochastic algorithms vs. systematic algorithms [Freuder *et al.*, 95]. Traditionally, constraint satisfaction algorithms relied on depth-first backtracking. These algorithms are systematic because of the following two properties [Pearl, 85]:

1. Do not leave any stone unturned (namely, they are complete).
2. Do not turn any stone more than once.

The unexpected efficiency of GSAT on difficult SAT problems [Selman et al., 92] raised the interest toward stochastic nonsystematic algorithms. These algorithms, inspired by physical and biological metaphors, consider constraint satisfaction as a cost function on the space of all total assignments which has to be globally optimized. They have the following properties (as Dechter indicates in [Freuder et al., 95]):

1. May leave many stones unturned.
2. May turn the same stone multiple times.

Since it was shown that stochastic algorithms can find solutions much faster than systematic algorithms for some interesting classes of problems, their intrinsic lack of completeness raised the following question: *if systematic algorithms may require more time than we can wait for, what do we want completeness for?* However, it was also shown that there are problem instances for which stochasticity perform poorly [Konolige 94]. In addition, because of their structure, stochastic algorithms are inappropriate for inconsistent problems.

A second topic generating the interest in the community was the evaluation of algorithms. The development of new systematic constraint satisfaction algorithms have been an intense topic of research for the last twenty years. By the time we started our work, many refinements to the basic backtracking strategy had been proposed. Some classical ones are: BM, BJ, GBJ, CBJ, FC, etc. and their combinations: BM-CBJ, FC-BJ, FC-CBJ, etc. [Prosser, 93a]. Some more recently developed are: MFC [Zweben and Eskey, 89; Dent and Mercer, 94], *weak-commitments* [Yokoo, 94], *nogoods-recording* [Frost and Dechter, 94], *dynamic-backtracking* [Ginsberg, 93], etc.

Their number had increased up to the point that it became important to order them according to their efficiency. A ranking based on a theoretical basis seemed out of reach. Since all algorithms have an exponential worst-case behaviour, ordering them according to their average-case performance on a benchmark was taken as the most reasonable approach. The problem was to find the appropriate benchmark. Previous evaluations were performed using *toy problems* such as the *n*-queens or the ZEBRA [Prosser, 93a], and it was accepted that any conclusion drawn from these experiments need not hold elsewhere because these *toy problems* are not representative of the problems that arise in practice. However, the discovery of the *phase-transition* on random problems and its associated average-case exponentially difficult problems provided a new source of problems to test algorithms on [Frost and Dechter, 94; Bacchus and van Run, 95]. It has to be accepted that random problems are not necessarily good representatives of real problems, but they provide an easy way to generate large samples of similar instances with an expected similar average-difficulty. The use of random problems has been a major step forward in constraint satisfaction benchmarking.

Experiments on random problems pointed that FC was one of the best algorithms for total constraint satisfaction. Enhancing it with intelligent backtracking (FC-CBJ) or lazy evaluation (MFC) only provided slight gains [Bacchus and van Run, 95]. The top position of FC in the algorithms' ranking reinforced the opinion that propagation is a key aspect on the effectiveness of algorithms. At this time it was believed that enforcing a lower level of propagation than arc-consistency was the most cost-effective overhead in constraint satisfaction [Kumar, 92]. When Sabin and Freuder [Sabin and Freuder, 94] suggested that maintaining full arc-consistency during search is often cost effective, the research community was skeptical about it.

General purpose heuristics for variable and value ordering have been an accompanying topic of research to that of systematic algorithms. Nevertheless, it has also attracted a lot of interest by itself. It has been shown for a long time that the order in which variables are instantiated strongly affect the size of the search space explored by algorithms. Dynamic orderings, where the current variable is selected at each node, are generally better than static orderings. More precisely, the minimum domain heuristic, already proposed in [Haralick and Elliot, 80] was believed to be the best variable ordering heuristic [Ginsberg et al., 90; Dechter and Meiri, 94; Bacchus and van Run, 95]. The question of choosing the value to use in the instantiation of the selected variable did not obtain as much attention as variable ordering. Nevertheless, research on value ordering was still taking part of major conferences [Geelin, 94; Frost and Dechter, 94]

Since machines become faster and more efficient algorithms are developed every day, constraint-based techniques can be applied to more and more real problems. When a real problem is cast in the CSP

framework, one often encounters overconstrained problems for which partial constraint satisfaction is of interest. Generalizing the CSP framework in order to deal with these situations was an intense topic of research that included the development of semantics in order to properly express these over-constrained real problems and their preference criterion [Shapiro and Haralick, 81; Schiex, 92; Martin-Clouaire, 92; Fargier and Lang, 93]. Not much work had been done on the algorithmic aspects of partial constraint satisfaction. The problem was addressed in [Freuder and Wallace, 92], where they generalize classical CSP algorithms to partial constraint satisfaction. This initial work pointed that the partial constraint satisfaction counterpart of FC was also the leading algorithm. Additional enhancements to partial constraint satisfaction algorithms included the development of MAX-CSP specific variable ordering heuristics [Wallace and Freuder, 93] and the use of directed arc-consistency pre-processing [Wallace, 94].

2.5.2 Research progress in 1995-98

The controversy between systematic and stochastic search is not over, yet. However, it is understood that each type of algorithms has its advantages and disadvantages. The suitability of each algorithmic schema probably depends on the domain where it is to be applied.

Regarding algorithms evaluation, random problems have become the standard benchmark for general purpose techniques. However, it is generally accepted that a final assessment of an algorithm requires some experiments on real domains [Frost *et al.*, 96]. Thus, it is becoming the norm that new algorithms for CSP be evaluated with both random problems and some specific domain. Especially relevant in this context is the recent work of [Kondrak and van Beek, 97] where, for the first time, some selected algorithms are theoretically ranked.

Regarding our own work, it started on the study of constraint satisfaction as global optimization. Our first idea was to apply local optimization techniques, such as the steepest ascent, to solve constraint satisfaction. Thus, it was close in spirit to that of stochastic search, but more influenced by hill-climbing than by stochasticity. Since hill-climbing does not guarantee global solutions, we decided to use hill-climbing advice inside systematic algorithms, by developing gradient-based variable and value ordering heuristics [Meseguer and Larrosa, 95; Larrosa and Meseguer, 95]. Interestingly, in the value ordering context, our intuitions drove us to very similar ideas of what was proposed in [Keng and Yun, 89; Geelen, 92; Frost and Dechter, 95]. However, we consider that our heuristics are more general because they have deeper foundations, include variable and value orderings in the same framework and can be applied to MAX-CSP. The next year (96), different variable ordering heuristics were

proposed [Bessière and Régin, 96; Gent et al., 96]. To date, no exhaustive evaluation of them has been reported.

When we were testing our heuristics on MAX-CSP, we rediscovered the use of DAC counts to improve PFC lower bound presented by [Wallace, 94]. However, our implementation was making more sensible use of the available information and provided better performance. These results were presented in [Larrosa and Meseguer, 96b]. As we were evaluating our algorithm on random problems, we found out that using DAC could change the difficulty pattern, so our algorithm (and Wallace's algorithm, too) could be exponentially better than plain PFC. This surprising result was presented in [Larrosa and Meseguer, 96a]. The same year, an alternative algorithm for MAX-CSP, *russian doll search*, was presented in [Verfaillie et al., 96]. Since we were not aware of each other's advances, no comparison could have possibly been done. However, it served to point out that many doors remain open in the MAX-CSP context. Also in that same year, Bessière and Régin provided convincing evidence of the importance of maintaining arc-consistency could have in difficult total constraint satisfaction problems [Bessière and Régin, 96]. Thus, our results, together with those of [Sabin and Freuder, 94] and [Bessière and Régin, 96] seem to be along the same line: enforcing local consistency is a promising approach for both CSP and MAX-CSP. Very recent work on both CSP [Debruyne and Bessière, 97] and MAX-CSP [Larrosa *et al.*, 98] give additional support to our claim. In [Meseguer and Larrosa, 97] we show that our techniques for MAX-CSP can be successfully adapted to more general frameworks of partial constraint satisfaction.

Nowadays, no one would doubt that MAC is the best algorithm for sufficiently hard problems in the total constraint satisfaction context and that PFC has been clearly outperformed by DAC-based PFC and PFC-RDS in the partial constraint satisfaction context. An exhaustive comparison between DAC-based PFC and PFC-RDS still remains to be done.

Lazy evaluation has been applied to constraint satisfaction in *Minimal Forward Checking* (MFC) [Zweben and Eskey, 89; Dent and Mercer, 94; Bacchus and Grove, 95] and in *Lazy Arc Consistency* [Schiex et al., 96]. In both cases, the corresponding non-lazy algorithms were doing more than needed to achieve their goals, so introducing this technique caused efficiency improvements. In that context, it was clear that the general idea of lazy evaluation was suitable in the constraint satisfaction context. Following that line of research, we have explored its applicability to the partial constraint satisfaction case and we have found that in this context lazy evaluation is even more suitable than in the total constraint satisfaction case. We believe that future efficient implementations of MAX-CSP algorithms will include some lazyness in their propagation procedure.

Finally, the idea of subproblem merging can be included into a line of research started with [Freuder, 91] where the first notions of value similarity were defined. The algorithmic implications of this concept

have motivated a sequel of results [Haselbock, 1993; Bellicha *et al.*, 94]. Our work on subproblem merging, presented in [Larrosa, 97], can be seen as a step forward a more flexible notion of value similarity with more practical applicability.

Chapter 3

Subproblem Merging

Classical CSP algorithms follow a common schema: at each node the algorithm selects a new variable and sequentally attempts to assign all its values. This schema is suitable for most problems. However, there are situations where it causes redundant search. In this Chapter we show one of these situations, which occurs when problems have different values for the same variable with a similar constraining behaviour. We show that the classical search schema does not take advantage of this similarity and solves the problem inefficiently because it considers these values as completely different. In this Chapter we develop one efficient way to deal with this situations. The idea is to transform the search space by merging sibling subtrees corresponding to similar values. As a result, we obtain narrower and higher search trees. We show that merging subtrees associated with similar values can reduce significantly the cost of search. This idea extends previous approaches with the same insight, in the sense that it is useful for a broader spectrum of situations. We develop algorithms for total and partial constraint satisfaction and experimentally show its suitability.

This Chapter is organized as follows. In Section 3.1, we introduce and motivate our approach. In Section 3.2, we review previous work related to our approach. In Section 3.3, we present the concept of value similarity. We introduce the concept of weak branching and show its merging effect in Section 3.4. In Section 3.5, we develop and analyze FCw, the FC extension to weak branching. The same idea is extended to MAX-CSP in Section 3.6 where we develop PFCw. In Section 3.7, we evaluate the validity of our approach on random problems and crossword puzzles. Finally, Section 3.8 contains some conclusions.

3.1 Introduction

In many real-world problems, variable domains may contain values which have, more or less, the same constraining behaviour. For example, consider a resource allocation task where a pool of resources have to be assigned to demanding jobs. Suppose that a given job requires a certain type of machinery, such as an oven. If there are a number of available ovens, each one with its own technical features, one can expect that expressing the problem as a CSP, different ovens having similar features will be associated with different values which may fit equally (or almost equally) well to problem solutions.

In tree search algorithms, each node represents a different subproblem defined by the assignments performed in the path from the root to the node. Two subproblems are *siblings* when they differ only in the value of the last assigned variable. Assigning values that have a similar constraining behaviour produces similar sibling subproblems. But depth-first based algorithms do not take advantage of this similarity because depth-first solves them independently. We claim that solving similar subproblems without exploiting their similarity is inefficient because part of the search is duplicated.

Figure 3.1 shows a simple graph coloring problem that illustrates this idea on total constraint satisfaction. The problem is to assign a color to each vertex in such a way that adjacent vertices have different colors. Color choices for each vertex are represented by letters and they appear inside the vertices. As CSP, vertices are problem variables, colors are values and arcs represent inequality constraints. Figure 3.1 shows the initial problem (upper left side) and the three sibling subproblems, S^a, S^b and S^c, that forward checking produces if X_1 is selected and its three values are assigned. Observe that they are not very different from each other. In particular, values a and b have a different effect in X_3 only, so assigning a or b to X_1 produces two similar problems that only differ in one value of X_3.

Depth-first based algorithms solve sibling subproblems sequentially and independently, without taking advantage of their possible similarity. These algorithms may repeat part of the search when solving different sibling subproblems. Figure 3.2 shows the search tree that forward checking traverses when it solves the coloring problem previously presented (the whole tree is traversed because the problem is unsolvable), assuming a lexicographical variable and value ordering. The two first subtrees, corresponding to S^a and S^b, have a large shadowed region formed by nodes with the same path, except in their first assignment. The reason for this is that values a and b produce two similar subproblems. Solving them independently (as forward checking does) causes a search repetition in both subproblems. The complete search requires FC to visit 24 nodes.

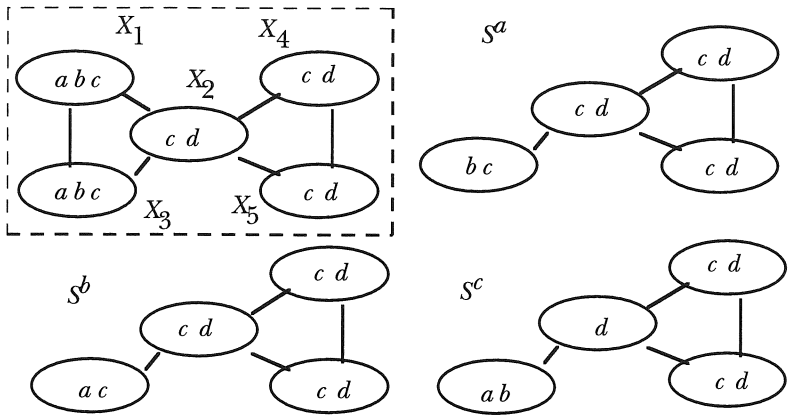


Figure 3.1: A coloring graph problem (left upper corner) and its three first-level subproblems.

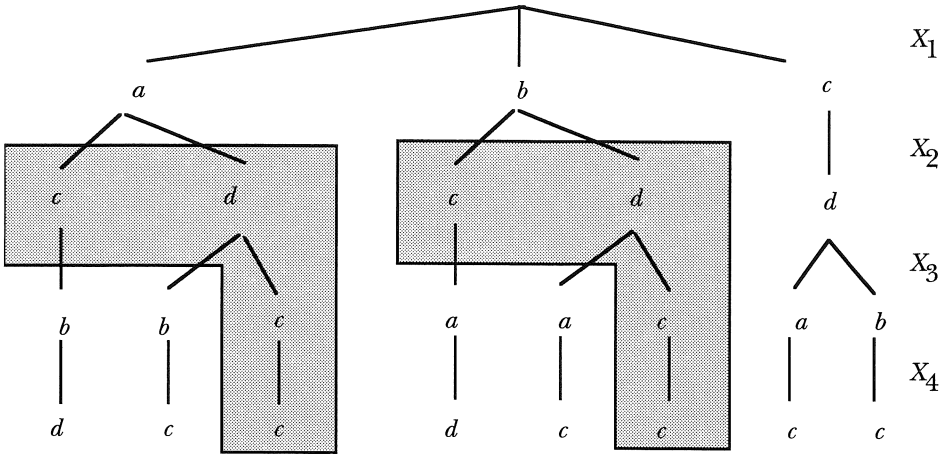


Figure 3.2: Search tree traversed by FC on the coloring problem.

In this Chapter we present a mechanism that is useful when dealing with problems having similar values. The basic idea is to group similar values of a variable (we say that they are weakly assigned) and to treat them as a single value. When all the variables have been assigned, the compound value is ungrouped. As a result, the associated subtrees are merged producing narrower search trees. The cost of this approach is a decrement in the algorithm's dead-end detection ability plus some extra levels in the tree, required when ungrouping compound values. Nevertheless, we show that if this idea is applied to sufficiently similar values, merging benefits surpass extra cost.

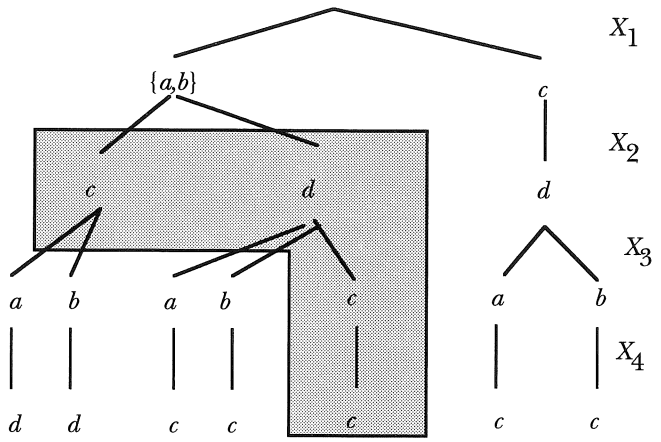


Figure 3.3: The merging effect of a weak assignment on the coloring problem.

Figure 3.3 presents the search tree that FC traverses if a and b values of X_1 are grouped in the coloring problem. Observe that pairs of duplicated nodes in Figure 3.2 are merged into a single node in Figure 3.3. In this particular case, the algorithm never reaches the point where the final decision about X_1 has to be taken because dead-ends are always detected before that point. The number of visited nodes is reduced to 19.

3.2 Previous Work

The importance of detecting values having similar constraining behaviour was first stressed by Freuder [Freuder, 91]. His work developed the concept of v*alue interchangeability*.

Definition 3.1:

Two values a and b of a CSP variable are *fully interchangeable* iff:

1. every solution to the CSP containing a remains a solution if a is replaced by b,
2. every solution to the CSP containing b remains a solution if b is replaced by a.

In other words, interchangeable values are indistinguishable with respect to the set of problem solutions. Interchangeable values are redundant in the sense that one of them can be removed without modifying the problem solvability. Moreover, the whole set of solutions can still be recovered. If only one representative of each group of interchangeable values is maintained, the problem space is simplified. Figure 3.4 shows a simple

graph coloring problem (adapted from [Freuder, 91]) that illustrates this idea. Colors c and d for vertex X_2 are fully interchangeable. For example, substituting c by d in the solution $\{X_1 \leftarrow a, X_2 \leftarrow c, X_3 \leftarrow a\}$ produces another solution.

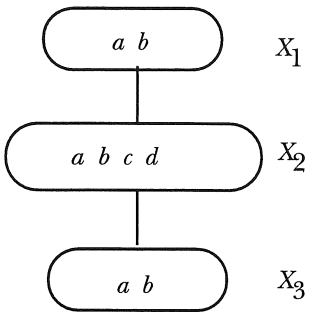


Figure 3.4: A graph coloring problem with interchangeable values.

Detecting fully interchangeable values does not seem to be an easy task. For this reason, this basic insight was extended to make it more useful in practice. Some forms of local interchangeability which are easier to detect are *neighbourhood interchangeability* and *neighbourhood substitutability*. They are easily defined in terms of the set of *supporting values*.

Definition 3.2:

Given a CSP and a value a of variable X_i, the *supporting values* of a in variable X_j are defined as,

$$supporting\_values(X_i,a,X_j)=\{b : \exists R_{ij} \text{ and } (a,b)\in R_{ij}\}$$

In other words, supporting values are the set of values a is consistent with. Consider now the following definitions of local interchangeability.

Definition 3.3:

Two values a and b of variable X_i are *neighbourhood interchangeable* iff for all variable X_j,

$$supporting\_values(X_i,a,X_j) = supporting\_values(X_i,b,X_j)$$

Neighbourhood interchangeability is held by those values with *exactly* the same constraining behaviour. It is of practical interest because it is a sufficient condition for full interchangeability that can be efficiently detected. In the example of Figure 3.4, colors c and d for vertex X_2 are indeed neighbourhood interchangeable. A somehow relaxed form of neighbourhood interchangeability is neighbourhood substitutability.

Definition 3.4:

Given two values a and b of variable X_i, value a is *neighbourhood substitutable* for b iff for all variable X_j,

$$supporting\_values(X_i,a,X_j) \supseteq supporting\_values(X_i,b,X_j)$$

Neighbourhood substitutability is *one way* neighbourhood interchangeability. It is of practical interest for two reasons: (*i*) it is a weaker condition, so it may happen more often in practice, and (*ii*) the value with smaller support can be removed without modifying the problem solvability. In the example of Figure 3.4, color c for vertex X_2 is neighbourhood substitutable for colors a and b.

Searching for one solution, values which do not modify the problem solvability can be safely removed. Neighbourhood interchangeability and substitutability give sufficient conditions for safe value removal and can produce important problem simplifications. Back to the example of Figure 3.4, we can remove value d because it is interchangeable with value c. We can remove values a and b because they are substitutable by value c. After the removal, values a and b of variables X_1 and X_3 become interchangeable, so value b can be removed from both variables. Consequently, we obtain a trivial problem which requires no search at all to be solved. Figure 3.5 shows the reduction process. It has been shown that detecting and exploiting local interchangeabilities at a pre-processing step is cost effective for some CSP.

With the aim of a more practical use of interchangeabilities, the concept of neighbourhood interchangeability is relaxed to be only applicable to a single constraint [Haselbock, 93].

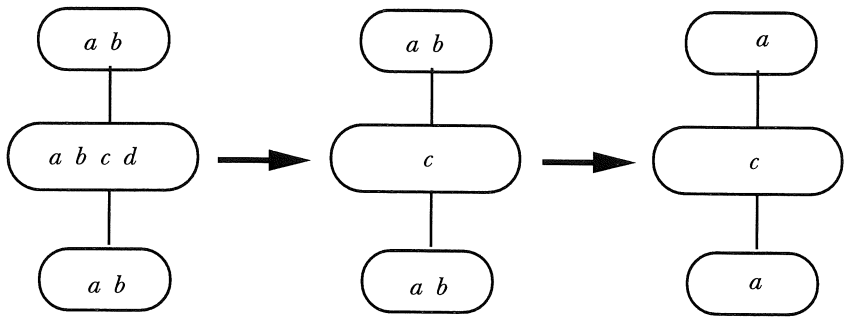


Figure 3.5: A graph coloring problem simplification by means of local interchangeabilities.

Definition 3.5:

Two values a and b of variable X_i are *neighbourhood interchangeable* with respect to variable X_j iff,

$$supporting\_values(X_i,a,X_j) = supporting\_values(X_i,b,X_j)$$

Considering interchangeabilities subject to a single constraint has the interest of providing weaker conditions that will happen more frequently in practice, as the following example illustrates.

Example 3.1: [Haselbock, 93]

Let variables X_1, X_2 and X_3 represent three ports of a board where modules must be mounted on. The available modules have two main characteristics: their mode (*analog* or *digital,* abbreviated as *a* and *d*) and their version number (1 or 2). Thus, domains of variables are $\{v_{a1}, v_{a2}, v_{d1}, v_{d2}\}$.

The following constraints restrict the possible configurations:

R_{12}: *the modules mounted on* X_1 *and* X_2 *must be of different modes.*

R_{13}: *the modules mounted on* X_1 *and* X_3 *must have different version numbers.*

From the perspective of X_1, the following assertions are true:

v_{a1} and v_{a2} are neighbourhood interchangeable with respect to variable X_2.

v_{d1} and v_{d2} are neighbourhood interchangeable with respect to variable X_2.

v_{a1} and v_{d1} are neighbourhood interchangeable with respect to variable X_3.

v_{a2} and v_{d2} are neighbourhood interchangeable with respect to variable X_3.

However, in this simple CSP there is no pair of neighbourhood or fully interchangeable domain values in X_1.

Following Haselbock's work, [Bellicha *et al.*, 94] uses the concept of local substitutability subject to a single constraint.

Definition 3.6:

Given two values *a* and *b* of variable X_i, value *a* is *neighbourhood substitutable* for *b* with respect to variable X_j iff,

$$supporting\_values(X_i,a,X_j) \supseteq supporting\_values(X_i,b,X_j)$$

Local interchangeabilities subject to a single variable can be detected in a pre-processing step with low computational cost. Once they are computed, they can be exploited during the subsequent search in two different ways [Haselbock, 93; Bellicha *et al.*, 94]:

1. *Consistency checks saving*: local interchangeabilities provide dominance or equivalence conditions which can be used to save consistency checks. For instance, in a FC-like algorithm, after checking an assignment $X_i \leftarrow a$ against a future value *b* of X_j, consistency with respect to locally interchangeable or substitutable values may be already known because of the pre-processing. Using this idea, no

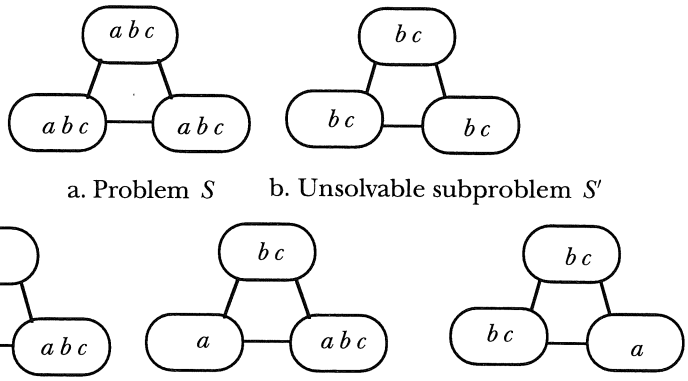
search tree reduction is obtained, but computational effort can be avoided, especially on those contexts where constraint checks are expensive.

2. *Subproblem reductions*: During search, algorithms face subproblems where the task consist on solving the original CSP subject to past assignments. It may be that some values were not interchangeable, nor substitutable at the initial problem, but become it at some subproblems. Then, the subproblem can be simplified by the appropriated domain reductions.

In [Haselbock, 93; Bellicha et al., 94] several algorithms based on this ideas are presented and evaluated. It is proven that they are cost-effective for some CSP.

Detecting local interchangeabilities is useful because they indicate value similarities which depth-first fails to exploit. However, local interchangeabilities are only some cases of flagrant value similarity. There are more subtle ways in which values can be similar. For instance, in the coloring problem of Figure 3.1, there is no pair of neighbourhood or fully interchangeable domain values. Nevertheless, it was shown in Figure 3.2 that FC repeats part of the search because S^a and S^b are similar. A different approach is followed in [Freuder and Hubbe, 95]. The idea involves the *extraction* from a problem of those subproblems that are known to be unsolvable.

Consider the coloring problems of Figure 3.6a and 3.6b. The second is a subproblem of the first (here we denote *subproblem* to one in which domains are proper subsets of the other). This particular subproblem is unsolvable. Obviously, it would be good to extract it from the original problem so no search is spent in proving its unsolvability. The following is the extraction procedure presented in [Freuder and Hubbe, 95]:



a. Problem S b. Unsolvable subproblem S'

c. The three components of S minus S'

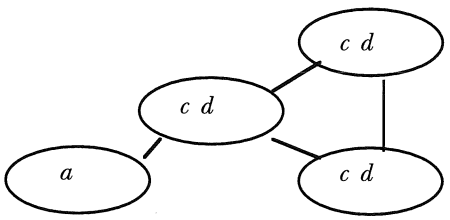
Figure 3.6. A decomposition example on graph coloring.

Extract (*Subproblem, Problem, Decomposition*)
Until the *Problem* matches the largest *Subproblem*
Pick a variable, X_i, whose domain at the *Problem* does not match its domain at the *Subproblem*
Divide the *Problem* into two subproblems that differ only in that the domain of X_i in one matches the domain of in the target *Subproblem*, while the domain of X_i in the other contains the remaining values
Set the *Problem* to the first of these subproblems
Add the second to the *Decomposition*
Apply Extract to the updated *Problem* and *Decomposition* with the same target *Subproblem*
Return the Decomposition

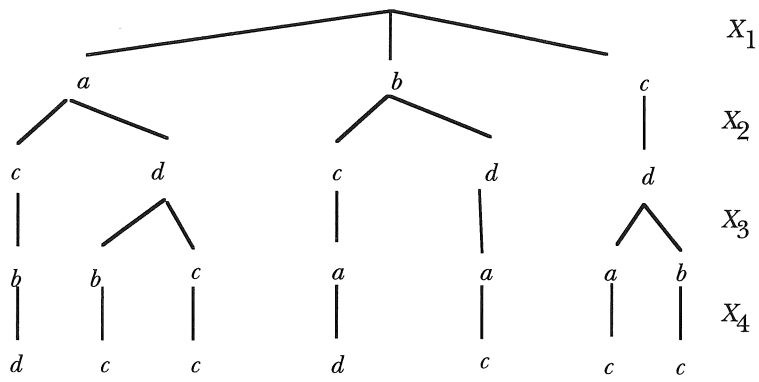
This procedure extracts *Subproblem* from *Problem*, producing a *Decomposition*. Figure 3.6c depicts the decomposition that is obtained if the subproblem of Figure 3.6b is extracted from the problem of Figure 3.6a. This extraction mechanism is used in a FC-like algorithm in the following way: before solving a subproblem, the common part that it has with the largest sibling problem that has been found unsolvable is extracted, producing a simpler problem.

Example 3.2:

Consider the coloring problem of Figure 3.1 and its three first level subproblems. It was shown that S^a and S^b have a good deal in common. After solving S^a and detecting its unsolvability, it is reasonable to extract from S^b the common part that it has with S^a(namely, S^b -($S^a \cap S^b$)). The extraction produces the following subproblem,



If FC solves the sequence S^a , S^b -($S^a \cap S^b$) and S^c, the following search tree is obtained,



Which requires visiting 22 nodes. Comparing this tree with the tree produced by the weak assignment of $\{a,b\}$ to X_1 (Figure 3.3), one observes that, for this particular case, a weak assignment produces a smaller search tree than a subproblem extraction.

3.3 Value Similarity

We say that two values of the same variable are *similar* when they are inconsistent with most of the same values of the problem. Thus, similarity between values is a *graded concept* based on the intersection of their supporting values. The more common values they have in their supports, the more similar they are. Our definition of similarity is deliberately vague, since we want it to be applicable over a broad class of situations.

Local interchangeabilities, in the sense of [Freuder, 91], are particular cases of our idea of similarity. Local interchangeable values have indeed maximal similarity, since they are inconsistent with exactly the same values. Local substitutable values have a special type of similarity in which one value dominates the other. However, they require strict conditions that may not occur very often in practice.

For example, consider a scheduling problem where the starting time of a set of jobs has to be decided. Consider the CSP representation in which jobs are variables and feasible start times are their values. For a given job, assigning its start time to t or t+1 will produce two subproblems that will not differ much from each other (they will probably only differ in that the second leaves one more period of time to preceding jobs and substracts one period of time to successors jobs). Thus, t and t+1 are similar values in a more subtle way that the one detected by local interchangeabilities. Our approach is appropriate for this more general idea of value similarity, where strict local interchangeability cannot be applied.

3.4 Weak Branching

Standard constraint satisfaction algorithms traverse a tree defined by the following branching rule. At node S an unassigned variable is selected. The successors of S are defined by instantiating the current variable to each value of its domain. We denote by S^a the successor of S in which value a is assigned to the current variable. We refer to this strategy as *strong branching* and we say that each subproblem is obtained after the *strong assignment* of the current variable.

Using strong branching, there is an equivalence between nodes, paths and assignments. Throughout this Chapter, unless otherwise indicated, we assume that algorithms select variables in lexicographical order. Thus, a node at level h is defined by its path from the root which is a partial assignment $\{X_k \leftarrow v^k: 1 \leq k \leq h\}$ (recall that v^k denotes the value assigned to X_k). We refer to paths under the strong branching strategy as *strong paths.*

The strong branching strategy is the responsible for the inefficient treatment that algorithms give to similar sibling subproblems: once a variable is selected, similar values produce similar subproblems that are solved independently. Our approach involves detecting these similar values and, when their variable is selected for instantiation, not to produce a different node for each similar value, but a unique node that includes all of them. The decision of what value is finally assigned to that variable is delayed to later tree levels. As a consequence, all search performed from this point to the final value splitting is common to all similar values.

To develop this idea we need to weaken the branching strategy and, instead of generating successors by selecting single values for the current variable, to allow the selection of groups of values. It requires the following definition.

Definition 3.7:

Given a node S, its current variable X_i and a subset of its domain $Q \subset D_i$ ($|Q|>1$), the *weak assignment* $X_i \leftarrow Q$ produces a successor node S^Q, such that in the subtree below only values in Q are considered for X_i.

The use of weak assignments allows for the definition of the *weak branching* strategy.

Definition 3.8:

Given a node S and a partition of its current variable domain, $\Pi=\{\pi_k\}$, the *weak branching* strategy defines the set of successors of S as the set $\{S^{\pi k}: \pi_k \in \Pi\}$ where $S^{\pi k}$ is the node associated with the

assignment $X_i \leftarrow \pi_k$. Assignments can be either strong or weak, depending on the cardinality of π_k.

When every subset in the partition has one element, weak branching becomes strong branching. It is easy to see that weak branching is a correct branching strategy because: (*i*) each successor subproblem is smaller than its parent, so all paths are finite, and (*ii*) no problem solutions are lost in the transformation. However, using weak branching, the algorithms traverses quite a different search tree. Paths may have different nodes associated with (weak) assignments of the same variable. Consequently, leaves occur at different tree levels, depending on the sequences of assignments which define their path.

For our purposes, it is enough to allow weak branching in a limited way. We only allow each variable to be assigned at most twice throughout a path. The first assignment can be either weak or strong and, if it is weak, there is a second strong assignment in which a single value for the variable is finally decided. Using this strategy the concept of path is no longer equivalent to a partial strong assignment.

Definition 3.9:

A *weak path* of length h ($h \leq 2n$) is a set of assignments, $\{Xs_k \leftarrow Q_k: 1 \leq k \leq h\}$, such that:

- The first n assignments represent strong or weak assignments to each problem variable.
- The remaining $(h\text{-}n)$ components represent strong assignments to the variables that were weakly assigned earlier in the path.

We extend the lexicographical variable ordering assumption to weak assignments. Thus, if $k \leq n$, Q_k represents an assignment (either weak or strong) to X_k. If $n < k \leq 2n$, Q_k represents a strong assignment to the $(k\text{-}n)$-th weakly assigned variable.

Therefore, under the weak branching schema search trees have variable height (bounded by $2n$) and two distinguishable areas:

- *Compression area*: up to level n, nodes represent search states to which strong and weak assignments are performed. This area is narrower than the corresponding strong branching search tree, because weak assignments have fewer successors than strong assignments.
- *Expansion area*: because of weak branching in the n first levels, the addition of new tree levels is required. From level n to level $2n$, nodes represent search states where some weak assignments done previously are made strong.

We can see the compression area of a weak branching tree as the fusion of some paths in its corresponding strong branching tree. This fusion is defined in the following.

Definition 3.10:

Given two search trees for the same problem, using strong and weak branching respectively, we define the application *fusion* from nodes of the strong branching tree to nodes in the compression area of the weak branching tree as follows,
$fusion(\{X_k \leftarrow v^k: 1 \le k \le h\}) = \{X_k \leftarrow Q_k: 1 \le k \le h\}$, such that $v^k \in Q_k$

The merging effect of a weak assignment, characterized by the *fusion* application, is illustrated in Figure 3.7. It sketches two search trees associated with the coloring problem of Figure 3.1. The upper tree is built under strong branching. The lower tree is built under weak branching, with a single weak assignment $X_1 \leftarrow \{a,b\}$. Observe that the first two subtrees of the upper tree are merged into a unique subtree in the lower tree. Moreover, an additional level is required in the bottom tree to assign a unique value to X_1 in the expansion area.

Observation 3.1:

The application *fusion* is exhaustive, but not injective. Each node in the compression area of the weak branching tree, defined by a weak path, $\{X_k \leftarrow Q_k: 1 \le k \le h\}$, merges $|Q_1| \cdot |Q_2| \cdot \ldots \cdot |Q_h|$ nodes.

Proof:

By definition of *fusion*, all nodes of the strong branching tree whose path is in the cartesian product of $Q_1 \times Q_2 \times \ldots \times Q_h$ correspond to a unique node $\{X_k \leftarrow Q_k: k \le h\}$ in the compression area of the weak branching tree. Obviously, there are $|Q_1| \cdot |Q_2| \cdot \ldots \cdot |Q_h|$ such nodes.

3.5 Application to Total Constraint Satisfaction

3.5.1 Forward Checking with Weak Assignments

Weak assignments can be easily embedded into algorithms for total constraint satisfaction adapting the algorithms to follow the weak branching strategy. With this approach, a search state has three different sets of variables: past variables (**P**) which have been strongly assigned, future variables (**F**) which have not been considered for instantiation so far and weakly assigned variables (**W**) which have been weakly assigned. If $\mathbf{F} \neq \emptyset$, the current node belongs to the compression area of the search tree. If $\mathbf{F} = \emptyset$, the current node belongs to the expansion area.

We have adapted forward checking to include weak assignments. The resulting algorithm, FCw, is outlined in Figure 3.8. FCw distinguishes two cases:

- If the set of future variables is not empty (*i.e.*: the current node is in the compression area), the current variable is selected among them (line 6). Its domain is partitioned and each subset is sequentially assigned. Unit subsets are strongly assigned (line 15) and non unit subsets are weakly assigned (line 18). Depending on the kind of assignment performed, the current variable is added to **P** or **W**. Each assignment is propagated with the weak look-ahead function. If the propagation does not produce an empty domain in **F** or **W**, the recursive call is made and search continues below.
- If the set of future variables is empty (the current node is in the expansion area), the current variable is selected among weakly assigned variables (line 24). Each value in its domain is strongly assigned (line 29). Each assignment is propagated with the weak look-ahead function. If no dead-end is detected, the recursive call is made.

The *weak_look_ahead* function propagates the effect of each assignment toward future and weakly assigned variables. If the current assignment is

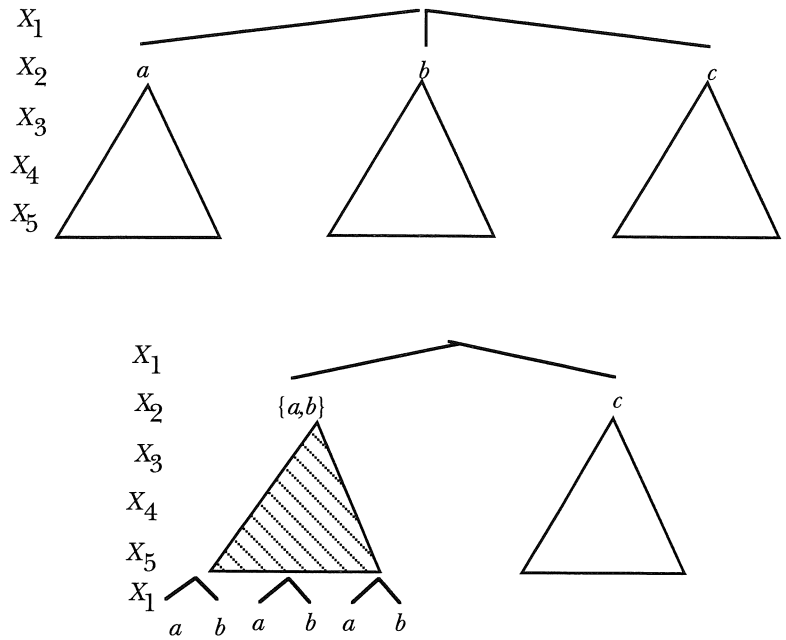


Figure 3.7: The merging effect of a weak assignment.

strong, look-ahead behaves as in the FC case, removing from domains of future and weak assigned variables those values that are inconsistent with the current assignment. If the current assignment is weak, look-ahead removes from non-past domains those values that are inconsistent with *all the weakly assigned values* (line 39). It is clear that a weak assignment has a lower pruning power than the strong assignment of any of its elementary values. Thus, FCw has a lower dead-end detection power than FC.

The *partition_domain* function chooses what values are weakly assigned. It must partition domains into groups of similar values subject to the current subproblem. This function is domain dependent (see Section 3.7).

3.5.2 Discussion on FCw

Weak assignments have been motivated as a way to merge search subtrees corresponding to similar values. In this section we show that, if FCw performs weak assignments of similar values, it can outperform FC. In our analysis, we compare the number of nodes visited by FCw and FC. Since both algorithms perform a similar amount of work at each node, the number of visited nodes can be considered a reasonable parameter to evaluate search effort.

Weak assignments modify the search tree structure by merging subtrees. In addition, weak assignments have the additional effect of modifying the relative order among leaves. Since algorithms stop when they find a solution, a weak assignment may affect their search efficiency by changing the relative position of the first solution leaf (for instance, if the first solution in lexicographical order is $\{X_1 \leftarrow b, X_2 \leftarrow a, X_3 \leftarrow a, .., X_n \leftarrow a\}$, a weak assignment $X_1 \leftarrow \{a, b\}$ changes its position from the $m^{(n-1)}+1$, to the second leaf). We are rather concerned with tree reductions. For this reason, in our analysis we consider unsolvable problems only. Doing so, we avoid the influence of the position of the first solution in the nodes that the algorithm visits. This assumption is realistic when dealing with non trivial problems because, even when they are solvable, algorithms spend most of their time proving the unsolvability of subproblems (observe that, if an algorithm requires visiting v nodes to solve a problem, in $(v\text{-}n)$ visited nodes their corresponding subproblem is unsolvable).

The relation between nodes visited by FC and FCw is given in the following observation.

```
function FCw (P, F, W, Assg, Dom) returns boolean
1       if (F = ∅ and W = ∅) then
2               sol := Assg)
3               return (true)
4       endif
5       if (F ≠∅) then
6               (X_i,D_i):= select_current_variable_and_domain(F, Dom)
7               Π:= partition_domain(D_i)
8               stop:= false
9               while (Π≠∅ and not stop) do
10                      π:= select_current_class(Π)
11                      Π:=Π-{π}
12                      NDom:= weak_look_ahead(X_i, π, (F∪W)-{X_i}, Dom-{D_i})
13                      if (not empty_domain(NDom)) then
14                              if (|π|=1) then
15                                      NAssg:= Assg ∪ {X_i←a}
16                                      stop:= FCw(P∪{X_i}, F-{X_i}, W, NAssg, NDom)
17                              else
18                                      D_i:=π
19                                      stop:=FCw(P, F-{X_i}, W∪{X_i},Assg, NDom∪{D_i})
20                              endif
21                      endif
22              endwhile
23      else
24              (X_i,D_i):= select_current_variable_and_domain(W, Dom)
25              stop:= false
26              while (D_i ≠∅ and not stop) do
27                      a:= select_current_value(D_i)
28                      D_i:= D_i-{a}
29                      NAssg:= Assg ∪ {X_i←a}
30                      NDom:= weak_look_ahead(X_i, {a}, (F∪W)-{X_i}, Dom-{D_i})
31                      if (not empty_domain(NDom)) then
32                              stop:=FCw(P∪{X_i}, F, W-{X_i}, NAssg, NDom)
33                      endif
34              endwhile
35      endif
endfunction
function weak_look_ahead(X_i, π, Vars, Dom) returns set of domains
36      stop:= false
37      for all D_j ∈Dom while (not stop)do
38              for all b ∈D_j do
39                      if (∀a∈π inconsistent(X_i←a, X_j←b)) then D_j:= D_j-{b} endif
40              endfor
41              if(D_j= ∅) then stop:= true endif
42      endfor
43      return (Dom)
endfunction
```

Figure 3.8: FCw.

Observation 3.2:

1. If FC visits node S, then FCw visits its fusion node $fusion(S)$ in the compression area.
2. If FCw visits node S in the compression area, then FC does not necessarily visit any node in $fusion^{-1}(S)$.
3. FCw may visit nodes in the expansion area.

Proof:

1. When a node is visited by FC, none of its ancestors has produced an empty domain. Therefore, if FC visits $\{X_k \leftarrow v^k: 1 \le k \le h\}$, all future variables have at least one feasible value in their domain before look-ahead. Let α_j $(h < j \le n)$ be such a value. By definition, $fusion(\{X_k \leftarrow v^k: 1 \le k \le h\}) = \{X_k \leftarrow Q_k: 1 \le k \le h\}$, such that v^k belongs to Q_k. If α_j is consistent with $\{X_k \leftarrow v^k: 1 \le k < h\}$, then it is also consistent with $\{X_k \leftarrow Q_k: 1 \le k < h\}$. Therefore, FCw visits $\{X_k \leftarrow Q_k: 1 \le k \le h\}$ because assignments previous to X_h do not produce an empty domain.
2. Consider a 3-variable problem with two values per variable (*i.e.* $D=\{a,b\}$) and the following constraints: $R_{12}=\{(b,a),(b,b)\}$ and $R_{13}=\{(a,a),(a,b)\}$. Then, FC detects a deadend after assigning either a or b to X_1. However, FCw does not detect the deadend after assigning $X_1 \leftarrow \{a,b\}$, so FCw will assign X_2. Thus, FCw visits a X_2 node while FC does not visit any.
3. Consider a 3-variable problem with three values per variable (*i.e.* $D=\{a,b,c\}$) and the following constraints: $R_{12}=R_{23}=\{(a,a),(b,b)\}$ and $R_{13}=\{(a,b),(b,a)\}$. It is easy to see that FCw visits the node $\{X_1 \leftarrow \{a,b\}, X_2 \leftarrow \{a,b\}, X_3 \leftarrow a, X_1 \leftarrow b\}$, which belongs to the expansion area.

From (1) we see how FCw can produce node savings with respect FC. A node, $\{X_k \leftarrow Q_k: 1 \le k \le h\}$, visited by FCw may merge up to $|Q_1| \cdot |Q_2| \cdot \ldots \cdot |Q_h|$ nodes visited by FC. Thus, weak assignments are useful when they merge many nodes visited by FC. From (2) and (3), we see how FCw may visit more nodes than FC. Since weak assignments decrease the pruning power, it may turn out that FCw has to go to deeper levels than FC to detect a dead-end. Thus, weak assignments will provide little or no savings when *weak* pruning is much lower than the alternative strong pruning.

We can explain the different performance of FC and FCw as a trade-off between branching and pruning. FC and FCw branch at different problem alternatives under the strong and weak branching strategy, respectively. At each node look-ahead propagates the current assignment by pruning unfeasible values. Considering dead-end detection, it is beneficial to follow the strong branching strategy where each decision is as specific as possible because it allows a stronger propagation which

prunes more values and may anticipate the detection of dead-ends. The disadvantage of strong assignments is that the search tree width grows faster with respect to tree levels. Considering the number of visited nodes, a weak branching strategy makes more general decisions, so each subproblem merges several strong branching subproblems producing narrower search trees. The disadvantage of weak branching is a weaker propagation. Thus, one needs to go deeper in the tree to detect dead-ends. Nevertheless, when similar values are weakly assigned, they basically allow the same propagation as if they were strongly assigned. In that situation a weak assignment is useful because it merges subtrees without decreasing the algorithm pruning capability.

From this observation, we conclude that the potential efficiency of FCw depends on its ability to find the right values to be weakly assigned. Ideally, a weak assignment should be done on a subset of values such that FC would repeat search for all of them. Similar values are good candidates for weak assignments because they have similar pruning capability, so it is expected that they will mostly visit the same nodes.

Finally, we want to mention that assuming a static ordering for variable and value selection was required to analyze the merging effect of weak assignments. However, it is not an algorithmic requirement of FCw. This is especially important from a practical point of view because it is well known that dynamic variable and value heuristics are much more effective than static ones in the total constraint satisfaction context.

3.6 Application to Partial Constraint Satisfaction

3.6.1 Partial Forward Checking with Weak Assignments

Algorithms for partial constraint satisfaction also follow the strong branching strategy. They can be adapted to weak branching following an approach similar to the total constraint satisfaction case. In this Section we develop the MAX-CSP counterpart of FCw, called PFCw. It appears in Figure 3.9. For clarity purposes, we have removed some feasibility tests that PFC includes to avoid some consistency checks.

Like in the FCw case, PFCw distinguishes three sets of variables at each search state: past (**P**), future (**F**) and weakly assigned (**W**). If the set of future variables is not empty, the current variable is selected among them. Its domain is partitioned and each subset is sequentially assigned (either strongly or weakly). If no future variable is left, the current variable is

selected among weakly assigned variables and each value in its domain is strongly assigned.

Look-ahead propagates the effect of assignments toward future and weakly assigned variables. It updates IC counts that are used to compute the lower bound. PFCw uses the following lower bound,

$$distance + \sum_{j \in \mathbf{F} \cup \mathbf{W}} min_v\{ic_{jv}\}$$

which is a straightforward extension of PFC lower bound, with the only difference that the sum of minima is extended to weakly assigned variables. However, given that a variable can be first weakly assigned and later strongly assigned, and both assignments involve look-ahead, this may cause to record twice in some IC the same constraint violation, which would render the previous lower bound invalid. To prevent this, look-ahead is modified to increment IC when it does not produce a duplication. In PFCw, this is done using an array of booleans, *propagated*, which records IC increments associated with weak assignments. More precisely, if $propagated_{jbi}$ takes value *true*, it means that X_i has been previously weakly assigned (*i.e.*: $X_i \in \mathbf{W}$) and its propagation produced an increment of ic_{jb}. This data structure is maintained and used by the look-ahead procedures. PFCw is shown in Figure 3.9. It has two different look-ahead procedures presented in Figure 3.10: *strong_look_ahead* is used to propagate strong assignments and *weak_look_ahead* is used to propagate weak assignments. They work as follows:

- *strong_look_ahead:* After a strong assignment, $X_i \leftarrow a$, ICs of values belonging to future and weakly assigned variables are updated. Consider an arbitrary value b, of a non past variable X_j. In general, if this value is inconsistent with the current assignment, its IC is increased (line 5). However, it is not increased if that constraint violation is already included in some IC. There are two possibilities:
  - X_j was previously weakly assigned and the weak assignment was inconsistent with $X_i \leftarrow a$. Then, ic_{ia} already includes that constraint violation. We can detect that situation (line 2) because $propagation_{iaj}$ was set to *true* during the weak assignment of X_j.
  - X_i was previously weakly assigned and the weak assignment was inconsistent with $X_j \leftarrow b$. Then, ic_{jb} already includes that constraint violation. We can detect that situation (line 3) because $propagation_{jbi}$ was set to *true* during the weak assignment of X_i.
- *weak_look_ahead*: After a weak assignment, $X_i \leftarrow \pi$, only ICs of future variables are updated. If a future value is inconsistent with all values weakly assigned, its IC is incremented (line 19). This increment is safe because the weakly assigned variable was selected among future variables and ICs of future variables do not

```
procedure PFCw (P, F, W, dist, Assg, Dom, IC)
1       if (F = Δ and W = Δ) then
2               UB:= dist
3               Best_sol:= Assg
4       else
5               if (F π Δ) then
6                       (Xi,Di):= select_current_variable_and_domain(F, Dom)
7                       P:= partition_domain(Di)
8                       while (P π Δ) do
9                               π:= select_current_class(Π)
10                              Π:=Π-{π}
11                              if (|π|=1) then
12                                      NAssg:= Assg ∪ {X_i←a}
13                                      ndist:= dist + ic_ia
14                                      (NDom,NIC):=
15                                              strong_look_ahead(ndist, X_i, π,
                                                        F-{X_i}, W, Dom-{D_i}, IC)
16                                      if (not empty_domain(NDom)) then
17                                              PFCw( P∪{X_i}, F-{X_i},
                                                        W, ndist, NAssg, NDom, NIC)
18                                      endif
19                              else
20                                      (NDom,NIC):=
21                                      weak_look_ahead(dist, X_i, π, F-{X_i}, W,Dom-{D_i}, IC)
22                                      if (not empty_domain(NDom)) then
23                                              D_i:=π
24                                              PFCw(dist,P,F{X_i},W∪{X_i},dist,Assg,
                                                                NDom∪{D_i},NIC)
25                                      endif
26                              endif
27                      endwhile
28              else
29                      (X_i,D_i):= select_current_variable_and_domain(W, Dom)
30                      while (D_i ≠∅) do
31                              a:= select_current_value(D_i)
32                              D_i:= D_i-{a}
33                              NAssg:= Assg ∪ {X_i←a}
34                              ndist:= dist + ic_ia
35                              (NDom,NIC):=
36                                      strong_look_ahead(ndist, X_i, a, F,
                                                        W-{X_i},Dom-{D_i}, IC)
37                              if (not empty_domain(NDom)) then
38                                      PFCw(ndist, P∪{X_i}, F, W-{X_i},
                                                        ndist, NAssg, NDom, NIC)
39                              endif
40                      endwhile
41              endif
42      endif
endprocedure
```

Figure 3.9: PFCw.

```
function strong_look_ahead(ndist, X_i, a, F, W, Dom, IC)
1       stop:= false
2       for all X_j ∈(F∪W) while (not stop) do if (not propagated_iaj) then
3               for all b ∈D_j  do if (not propagated_jbi) then
4                       if (inconsistent(X_i←a, X_j←b)) then
5                               ic_jb:=ic_jb+1
6                               Vars:= (F∪W)-{X_j}
7                               if (ndist + ic_jb + Σ_{k∈Vars} min_v{ic_kv} ≥ UB ) then
8                                       D_j:= D_j-{b}
9                               endif
10                      endif
11              endfor
12              if (D_j = ∅) then  stop:= true endif
13      endfor
14      return (Dom, IC)
endfunction
function weak_look_ahead(dist, X_i, π, F, W, Dom, IC)
15      stop:= false
16      for all X_j∈F while (not stop) do
17              for all b ∈D_j  do
18                      if (∀a∈π inconsistent(X_i←a, X_j←b)) then
19                              ic_jb:=ic_jb+1
20                              propagated_jbi:=CIERTO;
21                              Vars:= (F∪W∪{X_i})-{X_j}
22                              if (dist + ic_jb + Σ_{k∈Vars} min_v{ic_kv} ≥ UB) then
23                                      D_j:= D_j-{b}
24                              endif
25                      endif
26              endfor
27              if (D_j = ∅) then stop:= true endif
28      endfor
29      return (Dom, IC)
endfunction
```

Figure 3.10: Look-ahead procedures, as needed in PFCw.

have any contribution from constraints of other future variables. If a given IC is incremented, say ic_{jb}, then $propagated_{jbi}$ is set to *true* (line 20). It records that ic_{jb} has a contribution of a detected constraint violation between variables X_i and X_j. ICs of weakly assigned variables are not updated because X_i may have in its ICs contributions of them.

Like in the FCw case, the *partition_domain* function chooses those values to be weakly assigned. It should partition domains into groups of similar values subject to the current subproblem. Unlike FCw, similarity with respect to past variables is now required because different values may have a different level of consistency with respect to past variables. The actual implementation of this function is domain dependent.

Finally, we want to mention that PFCw can be combined with static and dynamic heuristics for variable and value selection.

3.6.2 Discussion on PFCw

PFCw solves MAX-CSP traversing a weak branching tree. Thus, each node visited by PFCw in the compression area of the tree merges a number of nodes that PFC may require to visit. However, each node visited by PFCw has a worse lower bound than its corresponding nodes in the strong branching tree, and therefore, it has a weaker dead-end detection. This is because of the following two reasons:

- When a weak assignment is performed, the contribution of the current variable to the lower bound is the lowest IC among the weakly assigned values. If these values are strongly assigned, at each subproblem the current variable has a contribution to the lower bound greater than or equal to the minimum IC of the current variable.
- Weak assignments have a lower power to increase ICs than its corresponding individual strong assignments because ICs are only increased when values are inconsistent with all weakly assigned values. Thus, after propagating a weak assignment, the minimum IC of each non past variable is lower than or equal to the minimum IC of each non past variable if the values are strongly assigned.

Therefore, PFCw visits less nodes than PFC when the merging produced by weak assignments outweighs the additional nodes that it visits because of its poorer dead-end detection. As in the FCw case, similar values with respect to future variables are good candidates for weak assignments. Similar values are mostly consistent with the same values of other domains, so most IC that would be increased by each strong assignment are also increased by the weak assignment. Regarding past variables, it is enough to select values for weak assignment such that they have a similar number of inconsistencies so their minimum IC is not very different from their maximum. In summary, weak assignments grouping similar values with respect to future variables (and with close IC) are good candidates to merge subtrees without a high decrement in their dead-end detection power.

As in the total constraint satisfaction case, we can explain the different performance of PFC and PFCw as a trade off between branching and pruning. PFC and PFCw branch at different problem alternatives under the strong and weak branching strategy, respectively. At each branch, their look-ahead propagates the assignment by pruning unfeasible values and increasing IC. On the one hand, it is beneficial to follow the strong branching strategy because it allows a stronger propagation (*i.e.*

more IC are increased) which prunes more values and may anticipate the detection of dead-ends. The disadvantage of strong assignments is that the search tree grows faster with respect to tree levels. On the other hand, a weak branching strategy makes more general decisions, so each subproblem merges many subproblems obtainable with strong branching, producing narrower search trees. The disadvantage of weak branching is that propagation is weaker and one needs to go deeper in the tree to detect dead-ends. When similar values are weakly assigned, they basically allow the same propagation as if they were strongly assigned. In that situation a weak assignment is useful because it merges subtrees without decreasing the algorithm pruning capability.

3.7 Experimental Results

We have discussed that weak assignments are cost-effective when they group similar values where similarity depends on the constraining behaviour of the values. In this section, we make a practical evaluation of the effect of weak assignments on algorithmic performance. Our experiments are comprised of two domains: random problems and crossword puzzles.

3.7.1 Correlated Random Problems

Standard random problems are not appropriate to benchmark FCw and PFCw because their random nature and their lack of structure makes the existence of similar values very unlikely. For this reason, we have devised a different model of random problems where value similarity depends on a structural parameter which can be varied to obtain different classes of problems. We call this type of problems *correlated random binary problems.* We must mention here that this model is totally artificial and is probably useless out of the weak assignments context. However, it is a useful tool to illustrate the effect of weak assignments.

Correlated random binary CSP assume and ordered set of variables and values. They are defined by five parameters $<n, m, p_1, p_2, \rho>$. The number of variables is n. The number of values per variable is m, which we assume even. Each value is refered to by their index $\{1, 2,..., m\}$. Parameter p_1 is the probability of a pair of variables being constrained (graph connectivity). Parameters p_2 and ρ define the problem tightness and value similarity in the following way: Given a pair of constrained variables (X_i, X_j) such that $(i<j)$, p_2 is the probability of the pair of values (a,b) with odd a, being forbidden (*i.e.*: $(a,b) \notin R_{ij}$). ρ is the probability of the pair of values (c,d) with even c, having the same constraining behaviour as $(c\text{-}1,d)$.

Figure 3.11 shows how correlated random problems are generated. Observe that, if ρ is set to 1, pairs of consecutive values (odd, even) have exactly the same behaviour with respect to posterior variables (in lexicographical order). If ρ is set to 0, pairs of consecutive values (odd, even) have exactly the opposite behaviour with respect to posterior variables. Increasingly more similar pairs of values occur as ρ is increased.

We experimented on the class of <20, 6, 50, p_2, ρ> problems. Parameter ρ varied in steps of 1/10 in the range 0.6-1.0. Parameter p_2 varied in steps of 1/36 in the range 0-1. For each parameter setting, samples of 100 instances were generated.

We evaluated the effect of performing weak assignments on these problems in both total and partial constraint satisfaction. Regarding total constraint satisfaction, we assessed FCw vs. FC. Regarding partial constraint satisfaction, we evaluated weak assignments on a more elaborated version of PFC —called PFC-DAC[1]— that improves the basic version. PFC-DAC is described in Chapter 4. Its refinement including weak assignments —denoted PFCw-DAC— is straightforward and does not need any description.

All algorithms tested in this and the following Chapters are programmed in C and compiled with *gcc*. The *-O3* optimization option is used in all cases. Looking for a fair performance comparison in terms of CPU time, all algorithms share code and data structures whenever it is possible. Algorithms are always executed on *Sun* Workstations either *Ultra1* or *Ultra2*. Different experiments may be carried out on different machines but, obviously, when CPU time is compared, the same computer is used. To measure CPU time, we use the *set_timers* and *elapsed_time* functions based in the well known [Manchak and van Beek, 94] CSP library. Finally, randomness is generated using the *lrand48* () function.

In these experimetns, all algorithms use lexicographical variable and value selection. Weak-assignment-based algorithms perform weak assignments of consecutive pairs of values of the form (*a*, *a*+1) where *a* is an odd number whenever possible (*i.e.*: both values are feasible at the time of the assignment).

Figure 3.12 reports results of executing FC and FCw. Five graphs show the search cost of solving the problems with the five different values of ρ. Each graph plots p_2 versus the average number of consistency checks for both algorithms. As was expected, FCw clearly outperforms FC on problems with high ρ. As ρ decreases, the gain given by FCw decreases as well. FCw seems to start losing its advantage for low tightness problems before it does for high tightness problems (see plot with ρ =0.7). There is a point, when ρ takes value around 0.65, after which weak assignments start being counter productive. In this set of experiments, at the problem class

[1] In fact, we are using PFC-DAC enhanced with the improvement presented in Section 4.4.

where FCw produces the highest gain in consistency checks, FCw is about 15 times better than FC in terms of checks.

Figure 3.13 gives results for the same experiment, but now reporting the number of visited nodes. A very similar behaviour can be observed, the only difference being that the gain ratio of FCw is greater in terms of nodes than in terms of checks. This fact is reasonable because weak assignments perform more checks per node than strong assignments (compare line 39 of Figure 3.8 with line 20 of Figure 2.5). Regarding nodes, FCw can be up to 80 times better than FC. Regarding CPU time, the gain ratio of FCw falls between the gain in terms of checks and the gain in terms of nodes.

```
procedure Generate_correlated_CSP (n, m, p1, p2, ρ)
1       for i:= 1 to n-1 do for j:= i+1 to n do
2               with probability p1 generate_constraint(i, j, m, p2, ρ)
3       endfor
endprocedure
procedure generate_constraint(i, j, m, p2, ρ)
4       for a:= 1 to m-1 in steps of 2 do
5               for b:= 1 to m do
6                       with probability p2 add (a,b) to Rij
7               endfor
8       endfor
9       for a:= 2 to m in steps of 2 do
10              for b:= 1 to m do
11                      with probability ρ add (a,b) to Rij if and only if (a-1,b)∈Rij
12              endfor
13      endfor
endprocedure
```

Figure 3.11: Correlated random problems generator.

Figure 3.14 reports results of executing the partial constraint satisfaction algorithms for ρ in the range 0.7-1.0. Again, each graph presents results for each different value of ρ. Each plot reports the average number of consistency checks on each problem class, for both algorithms. As in the previous case, PFCw-DAC outperforms PFC-DAC on problems with high ρ. As ρ decreases, the gain given by PFCw-DAC decreases as well. Again, tight problems seem to be more appropriate for our approach. Now the point after which weak assignments start being counter productive is around 0.75 for loose constraints and around 0.85 for tight constraints. In this set of experiments, at the problem class where PFCw+DAC produces the highest gain in consistency checks, PFCw+DAC is about 8 times better than PFC-DAC.

Figure 3.15 reports the number of visited nodes for the same experiment. Again, there is a strong correlation between checks and nodes, although weak assignments provide greater gains in terms of nodes. Regarding nodes, PFCw-DAC can be up to 40 times better than its competitor.

These experiments confirm that our approach is suitable on problems having sufficiently similar values, whose similarity can be detected. Moreover, we observe that the *degree of necessary similarity* for weak assignments to pay off is reasonable. Moreover, they give insight on the gain that one may expect. From the experiments, we also extract that performing weak assignments to insufficiently similar values can be highly counterproductive.

3.7.2 Crossword Puzzles

In the previous Section, we observed that FCw and PFCw can outperform FC and PFC in problems having sufficiently similar values. However, correlated random problems are unrealistic and do not show the real applicability of our approach. For this reason, we performed additional experiments on a domain where value similarity is both common and easily detectable. Crossword puzzles are a class of CSP that fulfil these requirements. In addition, they have already been used as benchmark in the CSP context [Ginsberg et al., 90].

A *crossword puzzle* is defined by: (*i*) a set of variable length words that we call *dictionary* and (*ii*) a two-dimensional *grid* such that each of its cells is either white or black. Each consecutive sequence of white cells in the grid (either horizontal or vertical) defines a *slot*. Thus, each white cell belongs to exactly two slots. The problem consist in assigning dictionary words to horizontal slots in such a way that vertical slots also form dictionary words.

A crossword puzzle can be expressed as a binary CSPs where variables correspond to grid slots. For each slot, its domain is the set of dictionary words that have the appropriate length. Intersecting pairs of slots define constraints. Each constraint restricts valid combinations of words to those that have the same letter in the intersecting cell.

The crossword puzzle domain is an interesting benchmark because of the following reasons,

- Its formulation is simple yet the problem is challenging.
- It is easy to obtain or generate large sets of different problem instances.
- Although being an academic problem, it is of practical interest because it can be seen as an abstraction of configuration problems where one has to decide the components of an assembly fulfilling interdependency restrictions.

An interesting feature about crossword puzzles is that constraints are implicit in the letters forming the words. More than that, each letter defines the constraining behaviour of one constraint. Thus, regarding value similarity, two different words are exactly equivalent with respect to intersecting slots associated with cells where they have a matching letter and they are completely different with respect to intersecting slots

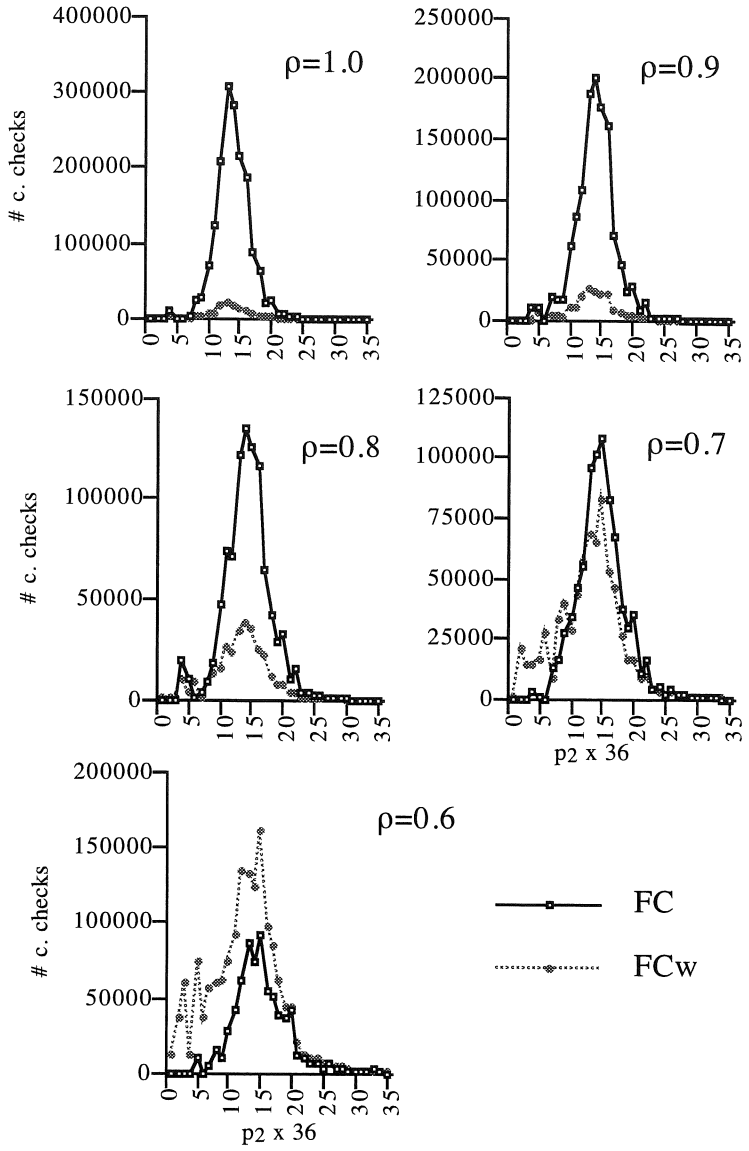


Figure 3.12: Average number of consistency checks of FC and FCw on five classes of correlated random problems.

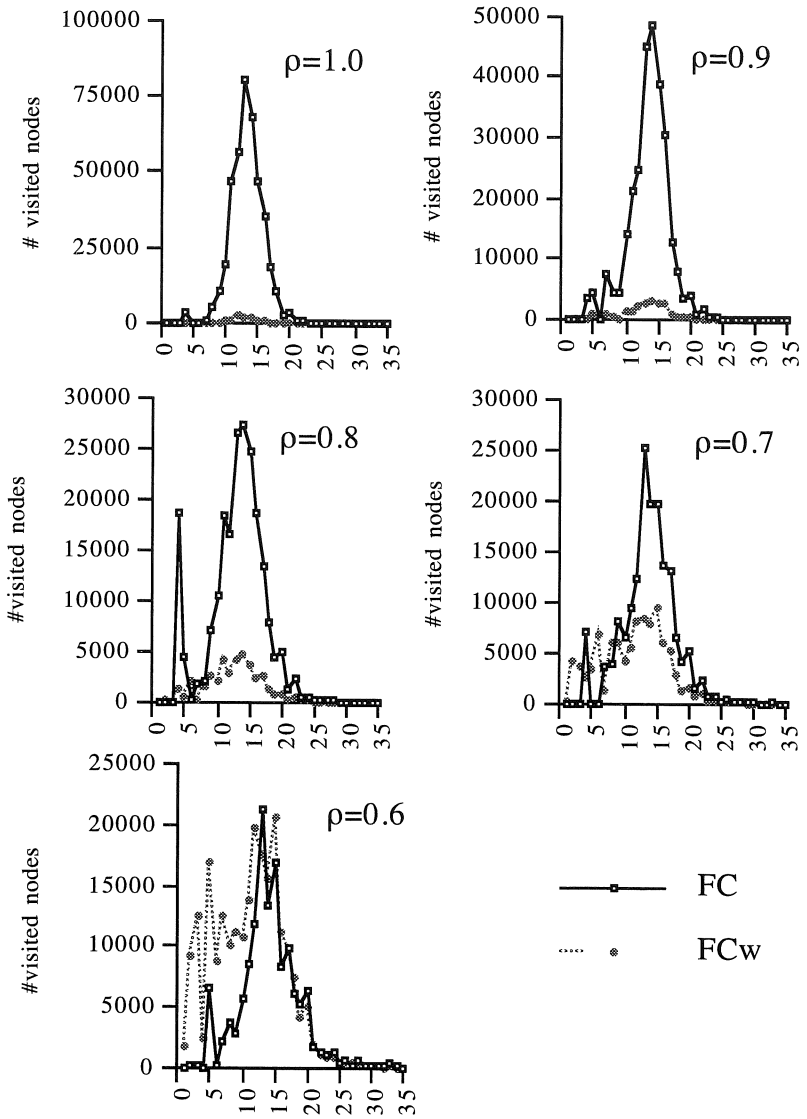


Figure 3.13: Average number of visited nodes of FC and FCw on five classes of correlated random problems.

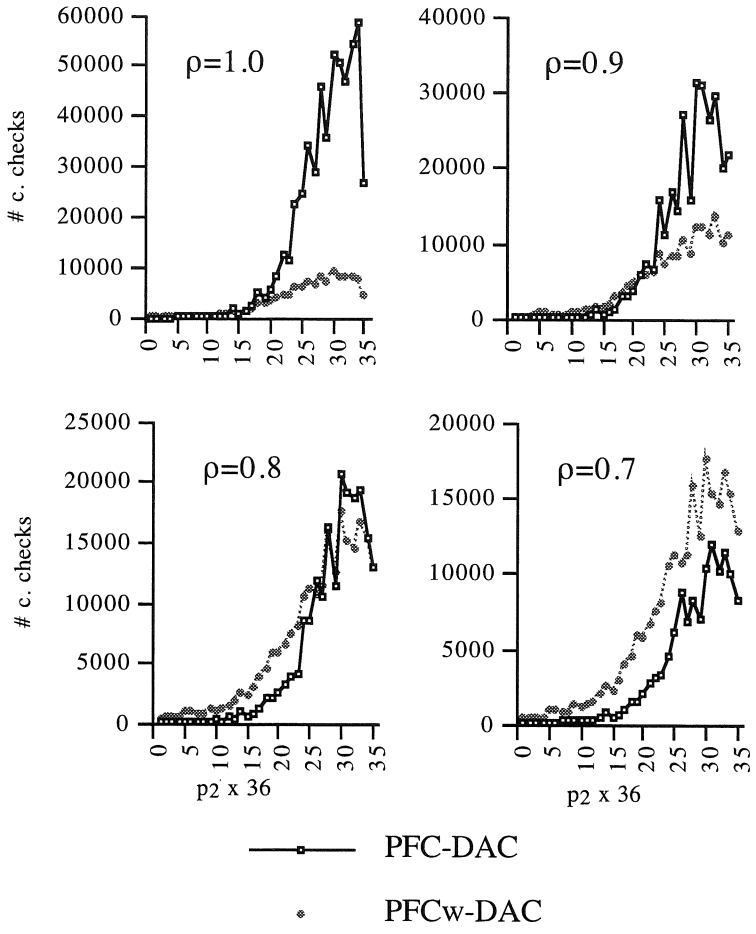


Figure 3.14: Average number of consistency checks of PFC-DAC and PFCw-DAC on five classes of correlated random problems.

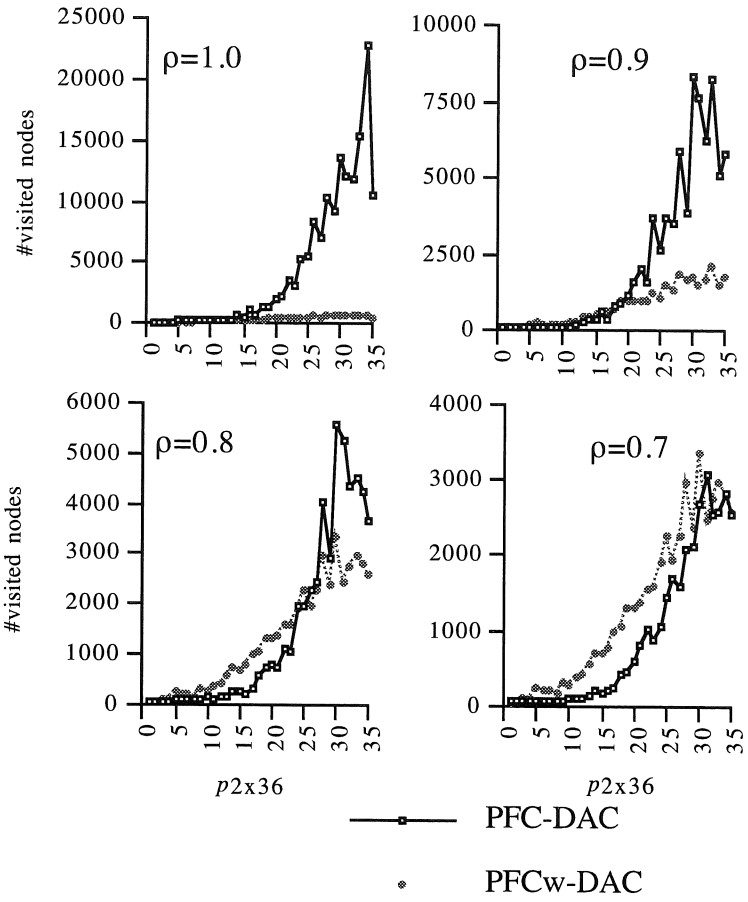


Figure 3.15: Average number of visited nodes of PFC-DAC and PFCw-DAC on five classes of correlated random problems.

associated with cells where they have a non matching letter. For example, consider a 6×6blank grid. We give consecutive variable names to horizontal slots in a top-down order (*i.e.*: X_1,..., X_6) and vertical slots in a left-to-right order (*i.e.*: X_7,..., X_{12}). Let us suppose that *castle*, *cattle* and *houses* are three words from the dictionary. Regarding their assignment to X_1, *castle* and *cattle* only have a different behaviour with respect to X_9 (Figure 3.16). Thus, their corresponding subproblems will be very similar. On the other hand, *houses* has a completely different behaviour with respect to every constraining variable. Thus, its corresponding subproblem will not have anything in common with the others. This feature is especially appropriated for our approach because we can get an idea of how similar two values are just by checking the number of matching letters.

Crossword puzzles are also interesting because weak assignments have a clear interpretation in their domain as we show in the following example. Consider that the 6×6 blank grid example is being solved by forward checking. After assigning the word *castle* to X_1, the look-ahead procedure removes from the domains of vertical slots those words which do not start by the corresponding letter. Thus, the new subproblem is subject to a decision in which 6 cells have been decided in the grid. After assigning the word *cattle*, the new subproblem is subject to a decision in which 6 cells have been decided (5 of them with the same value as in the previous case). However, after a weak assignment, $X_1 \leftarrow \{castle, cattle\}$, only 5 cells have been decided, the remaining cell has reduced its set of possibilities to the pair *s/t*. Thus, if we continue the search after the weak assignment, as long as we do not assign X_9, all search states are either valid for $X_1 \leftarrow castle$ and $X_1 \leftarrow cattle$. Therefore, each strong assignment fixes, in one go, all cells of the slot; but each weak assignment fixes some cells of the slot and only reduces the set of possibilities for the rest of cells.

	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}
X_1	**C**	**A**	***S/T***	**T**	**L**	**E**
X_2						
X_3						
X_4						
X_5						
X_6						

Figure 3.16. *castle* and *cattle* have a similar constraining behaviour.

In our experiments, we used two differently generated types of problems: (*i*) random crossword puzzles, where each word is randomly generated using a given alphabet, and (*ii*) random English crossword puzzles, where words are randomly selected from an English dictionary.

3.7.2.1 Random Words

This set of experiments was performed using random word puzzles with square blank grids. We used a three parameter model, where each puzzle is characterized by a tuple <*l*, *m*, *v*> such that:

- l is the word length and the grid dimension,
- m is the dictionary cardinality,
- v is the alphabet cardinality.

Random instances were generated by selecting m words out of the v^l choices using a uniform probability distribution.

We run FC and FCw on the following problem classes: <5,20-100,5>, <7,20-100,3> and <9,15-31,2>. For each parameter setting, samples of 100 random instances were generated.

Both FC and FCw used the *minimum domain* heuristic for variable selection. Regarding value ordering, values were always selected in lexicographical order (when weak assignments are performed, the lexicographical order is extended to the domain partition).

In our implementation of FCw, we classified the dictionary into groups of similar words in a pre-processing step and use this partition in the subsequent search. Words were considered similar enough to be weakly assigned when they had *less than three different letters*. We arrived to this value by sampling the dictionaries and observing that less similarity gave a partition which was too coarse. When providing CPU time, we always include the time required by this pre-processing.

Using m as the varying parameter, we observed that problems with small m are overconstrained and do not have any solution. If m is increased, there is a point after which problems become abruptly underconstrained and the number of solutions grows very fast. This point corresponds to a peak in average problem difficulty similar to that observed in other domains [Cheeseman *et al.*, 1991] (see Section 4.9.1 for a more comprehensive description). In our three problem classes the rank for m was chosen to coincide with the peak. Therefore, our experiments were in *hard* instances having few solutions or none at all.

Figures 3.17, 3.18 and 3.19 show the results of the experiment on the three classes, respectively. For each class we provide three plots: one comparing the number of consistency checks, another comparing the number of visited nodes and another comparing CPU time. It can be observed that FCw outperforms FC in all problem classes and for all search effort measures. The ratio of improvement can be up to 2.2 with respect to

checks and up to 2.5 with respect to visited nodes and time. The gain seems to be slightly greater as the grid size grows and the alphabet cardinality decreases. This fact is understandable because as search trees grow in depth, weak assignments at high tree levels have a larger merging power. Observe that gains are greater in terms of visited nodes and CPU time than in terms of consistency checks. Regarding nodes, the reason is that weak assignments require a larger number of checks for their look-ahead. Regarding CPU time, the reason is that weak look-ahead checks consistency with all values weakly assigned in one go, without any overhead in between. So weak look-ahead in FCw can perform more checks per second than standard look-ahead in FC.

3.7.2.2 English Words

In our last experiment on crossword puzzles we use real English words. Words are taken from *WordNet* [Miller, 90], a well-known English dictionary used by the *natural language processing* research community. As in the previous case, we use square and blank grids. On this model, each problem is characterized by two parameters:

1. Grid size.
2. Dictionary cardinality.

A single instance is generated by randomly selecting words of the appropriated length from the dictionary.

We compared FCw vs FC on 4 × 4 puzzles leaving the dictionary cardinality as the varying parameter (for each cardinality we generated 50 instances). Like in the previous case, similar words were detected in a pre-processing step and this clustering was used during the subsequent search. Words with less that three different letters were considered similar enough for being weakly assigned. Both algorithms used the *minimum domains* variable ordering heuristic and selected values in lexicographical order.

In this domain we also observe an easy-hard-easy difficulty pattern where the hardest instances occur at the point where problems change from being solvable to unsolvable. Figure 3.20 reports average values of the three search effort parameters on the experiment. As in the previous case, FCw outperforms FC in all of them. The gain ratio of FCw is about 2.5 in terms of visited nodes, 2.0 in terms of CPU time and 1.6 in terms of consistency checks.

3.8 Conclusions and Future Work

For many problem areas it is a matter of fact that some domain values of a variable behave in the same manner. We have shown that it is

inappropriate to treat them as completely different values because it causes algorithms to make the same mistakes for all of them.

The notion of *local interchangeability* captures some of these value similarities. Nevertheless, it requires strong conditions of equivalence or dominance among values. Thus, algorithms exploiting local interchangeabilities may have a limited applicability in practice. We have presented an approach that is more general in the sense that it does not require similar values to accomplish any particular condition. The only requirement is that similarity is based on terms of constraining behaviour. For instance, in the crossword puzzles domain, no pairs of values are neither locally interchangeable, nor locally substitutable. However, we showed that some values are similar in our broader sense.

With our approach we show that a search space transformation is a suitable technique for dealing with value similarity. More precisely, since similar values produce similar subproblems and similar subproblems traverse similar trees, merging their trees causes the algorithm to traverse them, somehow, *simultaneously*. As a result, mistakes are not repeated. Experiments on random correlated problems and crossword puzzles (both with random and English words) showed the suitability of our approach.

We believe that our vague notion of value similarity can be characterized using some distance measurement (*i.e.*: taking the set of supporting values and considering common occurrences). If an effective distance measurement can be found and it can be computed in an efficient way, the decision of what values are weakly assigned can be done in a domain independent way. The exploration of these ideas is left for future work.

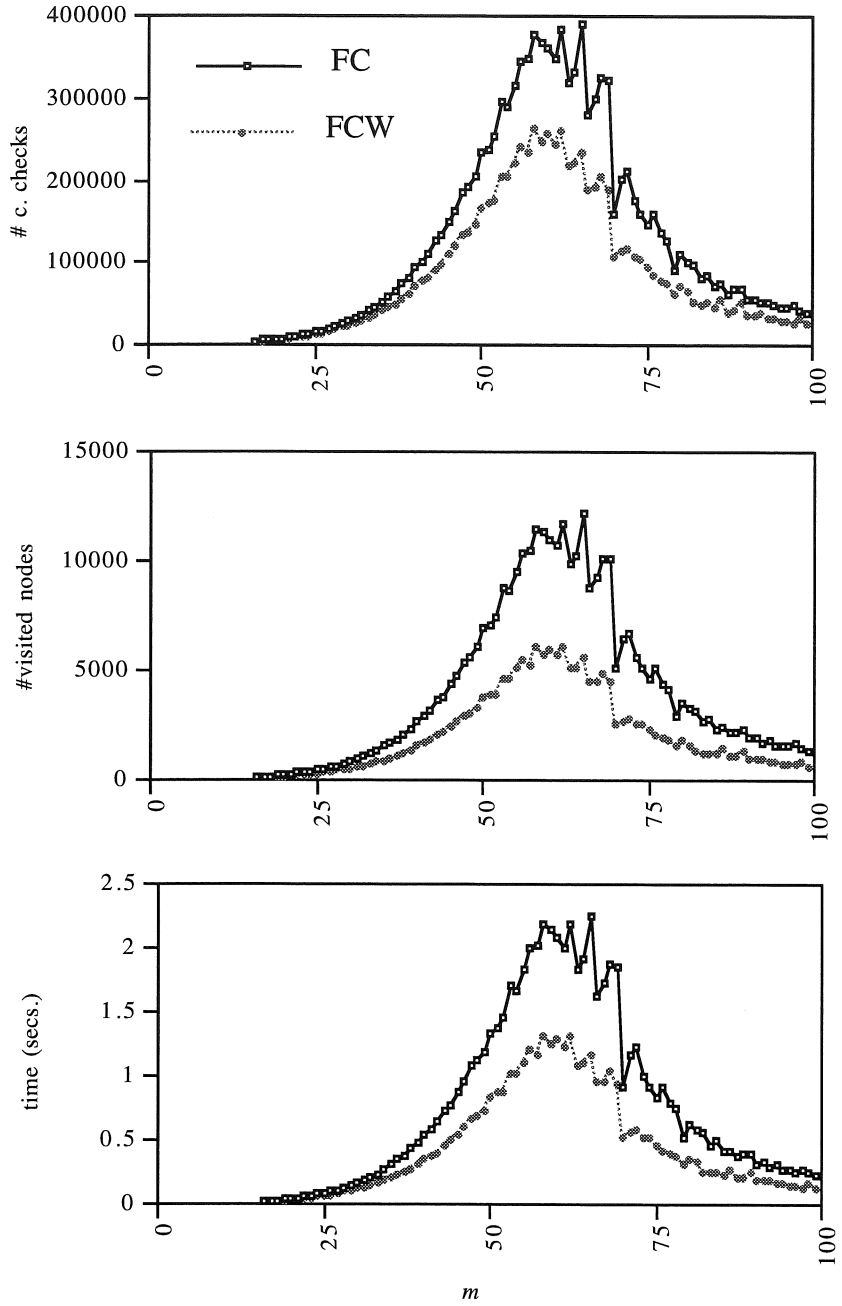


Figure 3.17: Experimental results on the <5, m, 5> class of random crossword puzzles. Each plot reports a different search effort measure.

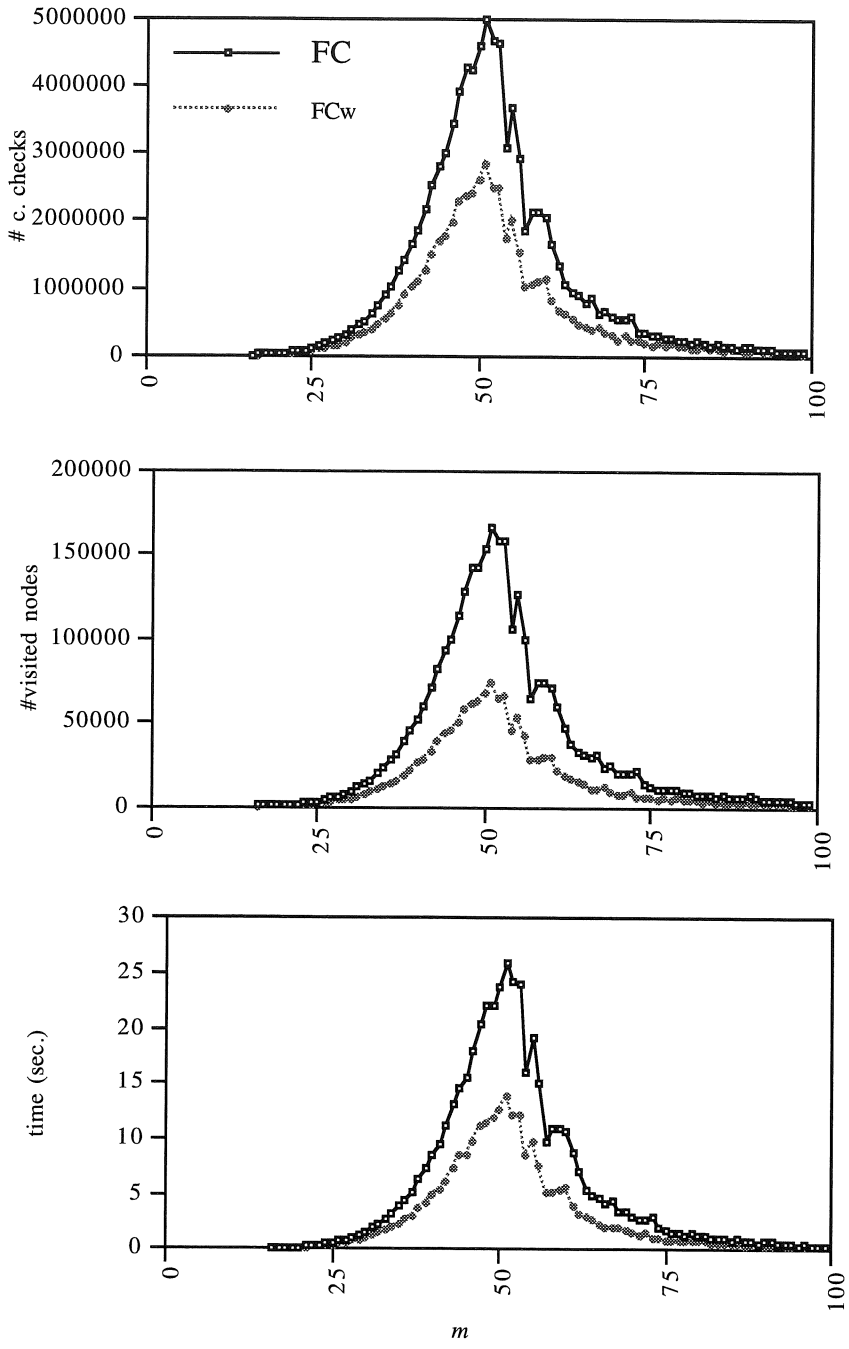


Figure 3.18: Experimental results on the <7, *m*, 3> class of random crossword puzzles. Each plot reports a different search effort measure.

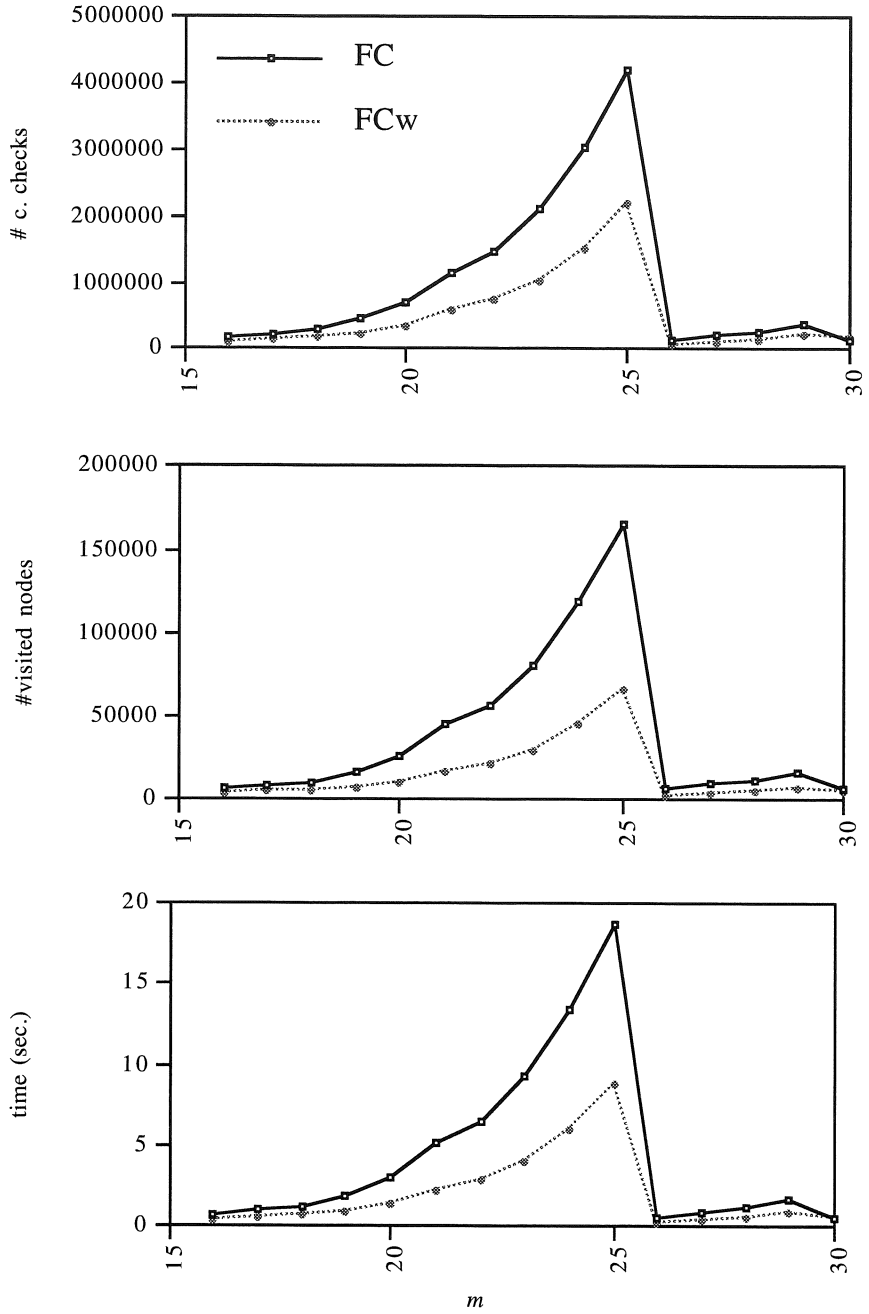


Figure 3.19: Experimental results on the <9, *m*, 2> class of random crossword puzzles. Each plot reports a different search effort measure.

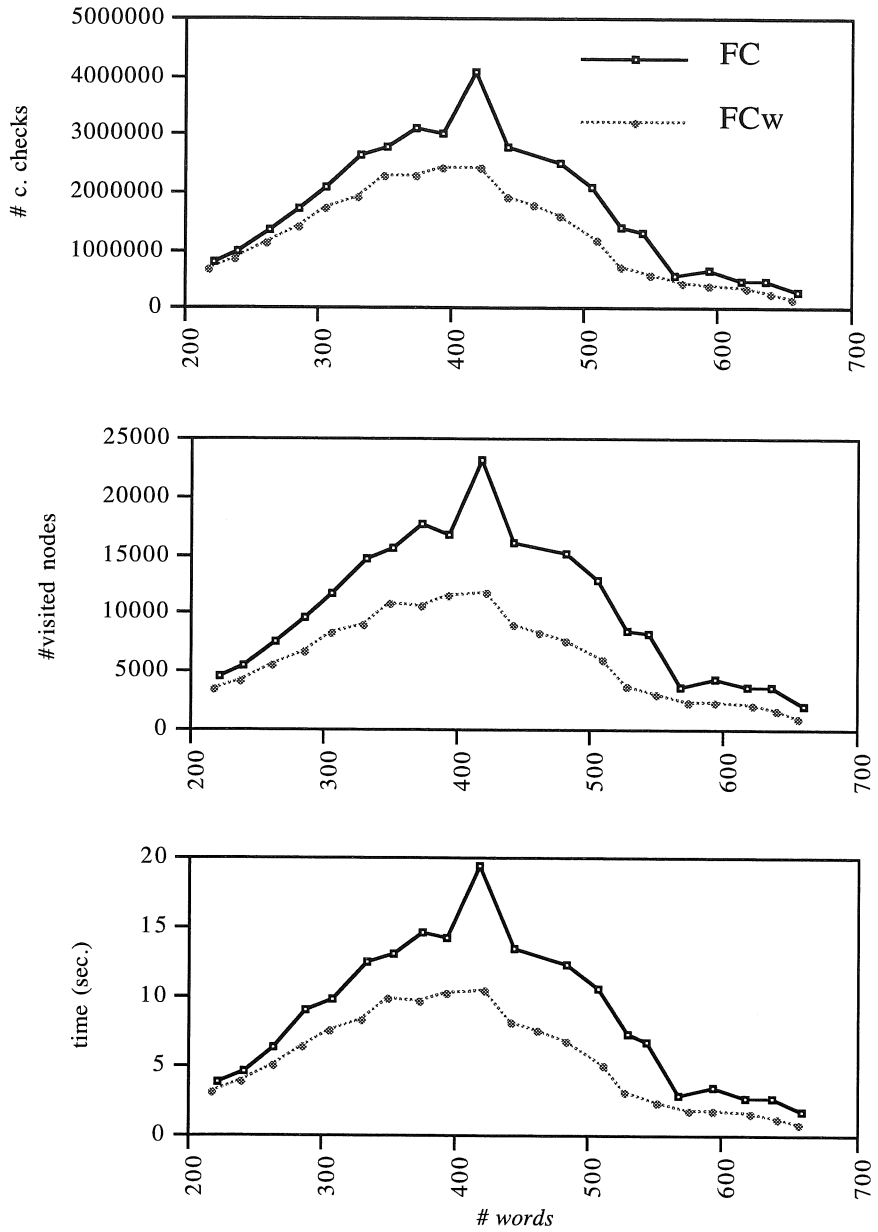


Figure 3.20: Experimental results on 4 x 4 blank grid crossword puzzles using random selection of English words from WordNet. Consistency checks, visited nodes and CPU time is reported.

Chapter 4

Combining Search with Local Consistency Enforcement in Partial Constraint Satisfaction

When search falls in a dead-end, it is condemned to traverse unsuccessfully the corresponding subtree. Therefore, it is of obvious interest to endow algorithms with good dead-end detection capabilities. In total constraint satisfaction, techniques for early dead-end detection involve combining search with local consistency. In this Chapter, we explore the suitability of the same idea in partial constraint satisfaction. In our approach, we detect constraint violations by means of arc-inconsistencies. This information is used to improve branch and bound lower bound. We present three new algorithms of increasing sophistication based on this idea which outperform their predecessors by several orders of magnitude. Additionally, we show that the use of local inconsistency information makes apparent a complexity peak in MAX-CSP not reported before. This complexity peak is unnoticeable when no local consistency information is used.

The structure of the Chapter is as follows. After an introduction in Section 4.1, we present previous work on the use of local consistency information to improve PFC in Section 4.2. In Section 4.3, we give some results which show the importance of using local consistency information in branch and bound. From Section 4.4 to Section 4.8 we introduce a sequence of incremental algorithmic improvements. In Section 4.9, we present and analyze the complexity peak of MAX-CSP and provide experimental support of the efficiency of our algorithms. Finally, in Section 4.10, we present the conclusions of the Chapter and point out some future work.

4.1 Introduction

Constraint propagation techniques are often used to detect and remove unfeasible values both during a pre-processing and during search. The basic propagation technique involves establishing some form of arc-consistency (Section 2.2.2). Arc-consistency is of interest in binary CSP because it removes values that cannot belong to any problem solution with low space and time requirements. Currently, alternative local consistency conditions for value removal are subject to research [Debruyne and Bessière, 97].

As we already explained in Section 2.2.3, forward checking is a very popular algorithm in constraint satisfaction. It combines backtrack search with a limited form of arc-consistency maintenance (at each stage, constraints between past and future variables are made arc-consistent). For many years it was believed that performing higher levels of propagation was not cost effective [Kumar, 92] and FC was considered the most efficient general algorithm. However, recent research on total constraint satisfaction has contradicted this belief. In [Sabin and Freuder, 94; Bessière and Régin, 96] there is strong experimental evidence of the important savings that maintaining full arc-consistency during search can produce.

In this Chapter we develop similar ideas for MAX-CSP. We give both theoretical and experimental evidence of the importance of using local consistency information, gathered during a pre-processing step, in branch and bound algorithms. In the MAX-CSP context, arc-inconsistent values cannot be removed because arc-inconsistency does not imply unfeasibility. Nonetheless, detecting an arc-inconsistent value indicates a necessary constraint violation associated with that value. Information about local inconsistencies can be used to compute better (higher) lower bounds during search. High lower bounds allow branch and bound to be more efficient by a better dead-end detection.

We continue the work started in [Wallace, 94], where the basis on how to combine PFC with local consistency information was given. Our contributions are twofold:

1. We provide theoretical evidence of the superiority of branch and bound algorithms using local consistency information. Roughly, we show that the polynomial overhead of these algorithms can produce exponential savings. More precisely, exponentially difficult problems for PFC, are trivially solved when arc-inconsistency information is added to its lower bound. Empirical examination of this fact has lead us to the discovery of a complexity peak on MAX-CSP not previously reported. We show that the search effort of PFC enhanced with the use of arc-inconsistency information presents an easy-hard-easy pattern in average difficulty when solving random binary CSP instances with increasing tightness. Interestingly, if the algorithm

does not use any local consistency information in its lower bound, the easy-hard-easy pattern does not occur and problems become increasingly hard. Surprisingly, the easy problems on the right part are the most difficult problems for plain PFC.

2. We present a set of improvements to the algorithm presented in [Wallace, 94] which lead to one of the best current algorithms for MAX-CSP. With our proposed improvements, a more suitable use of local consistency information is made which causes better lower bounds and, consequently, an earlier dead-end detection

4.2 Previous Work

As we already explained in 2.3.2, PFC is a forward checking algorithm for MAX-CSP. It uses look-ahead to propagate inconsistencies of previous assignments against future values. These inconsistencies are recorded in the so-called inconsistency counts (IC). Consequently, there is an IC for each future value recording the number of inconsistencies that it has with past assignments. These IC are used to compute branch and bound lower bounds. These lower bounds are used both to detect dead-ends and unfeasible values.

In the MAX-CSP context, arc-inconsistent values cannot be discarded because arc-inconsistency does not imply unfeasibility. But detecting arc-inconsistent values can still be useful because their arc-inconsistencies indicate necessary constraint violations associated with them. This information can be used to improve lower bounds. The first approach proposing the addition of arc-inconsistency information to the lower bound is due to [Wallace, 94]. Its approach is based on the concept of *directed arc-consistency*, first introduced by [Dechter and Pearl, 88] in the context of total constraint satisfaction (see Section 2.2.2). Directed arc-consistency is defined upon a total ordering among variables and allows for the characterization of so-called directed arc-inconsistency counts.

Definition 4.1:

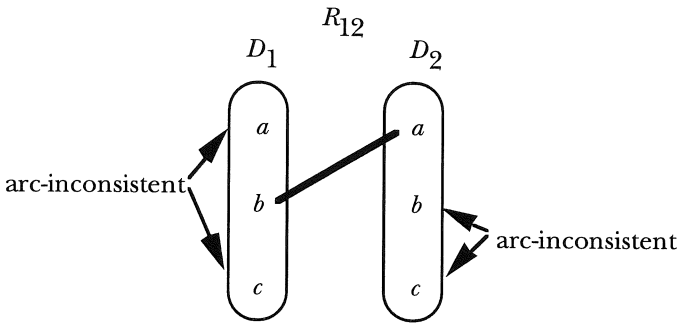
Given a CSP and an arbitrary but fixed variable ordering, the *directed arc-inconsistency count* (DAC) associated with value a of variable X_i, dac_{ia}, is defined as the number of variables which are arc-inconsistent with $X_i \leftarrow a$ and appear after X_i in the ordering.

For simplicity we assume the ordering to be lexicographical. Figure 4.1 shows the algorithm for DAC computation. For each variable X_i, the algorithm iterates on its posterior variables, checks its arc-inconsistencies and updates DAC.

Example 4.1:

Consider a CSP having four variables (*i.e:* $\{X_1, X_2, X_3, X_4\}$), three values per variable (*i.e:* $\{a, b, c\}$) and the following constraints: $R_{12}=\{(b,a)\}$, $R_{13}=\{(a,c), (b,c)\}$, $R_{14}=\{(a,a), (c,c)\}$, $R_{23}=\{(c,a), (c,b)\}$, $R_{24}=\{(a,a)\}$ and $R_{34}=\{(b,b), (b,c)\}$.

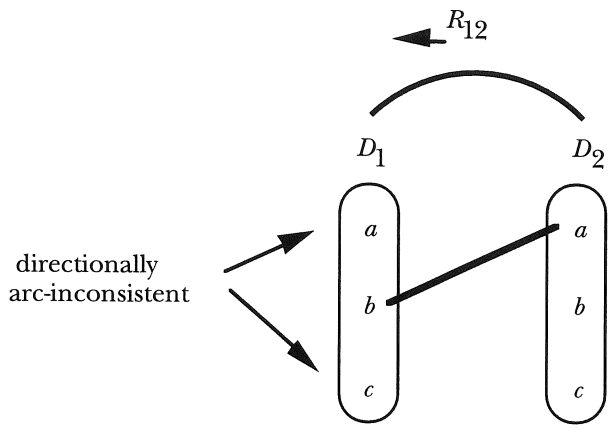
This problem has several arc-inconsistencies. For instance, regarding constraint R_{12} one observes that values a and c of X_1, and values a and b of X_2 are arc-inconsistent. What follows is a pictorical representation of the constraint where a solid line indicates the only compatible pair of values. Thus, values not having any solid line connecting them with the other variable do not have any consistent value, so they are arc-inconsistent.



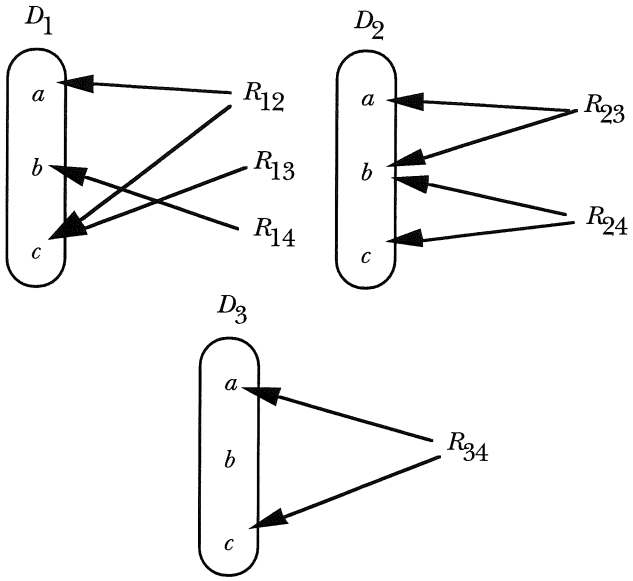
Arc-inconsistencies imply necessary constraint violations. Thus, from R_{12} one knows that any solution including $X_1 \leftarrow a$ will violate at least one constraint (R_{12}). The same reasoning can be done for $X_2 \leftarrow b$. However, constraint violations detected in that way cannot be added over variables because the sum can include the same constraint twice. Thus, from R_{12} one knows that any solution including $\{X_1 \leftarrow a, X_2 \leftarrow b\}$ violates *only one* constraint, although both values are arc-inconsistent.

One way to circumvent the constraint redundancy when adding arc-inconsistencies is to give an order to variables. This order induces a direction for each constraint. If only arc-inconsistencies subject to the constraint direction are considered, they can be added without any constraint duplication.

DAC counts are computed under a variable ordering. For each constraint only arc-inconsistencies of one variable contribute to DAC counts. Back to constraint R_{12}, only arc-inconsistencies of X_1 contribute to DAC counts.



The following picture illustrates individual directional arc-inconsistencies that contribute to DAC counts for this problem. Observe that each constraint only contributes to one variable counts (*i.e:* its first variable in lexicographical order). For this reason X_4 does not have any contribution. Each individual DAC is precisely the number of arrows pointing at its value (*i.e:* $dac_{1\,a} = 1$, $dac_{1\,b} = 1$, $dac_{1\,c} = 2$, *etc.*).



Directional arc-inconsistencies can be added over variables. Thus, any solution including $X_1 \leftarrow c$ will violate at least two constraint (R_{12} and R_{13}); any solution including $X_2 \leftarrow a$ will violate at least one

constraint (R_{23}); and any solution including $\{X_1 \leftarrow c, X_2 \leftarrow a\}$ will violate at least three constraints (R_{12}, R_{13} and R_{23}).

DAC can be used to detect necessary constraint violations between variables, as the following observation indicates.

Observation 4.1:

1. The DAC count associated with value $a \in D_i$, dac_{ia}, is a lower bound of the number of inconsistencies of any total assignment including $X_i \leftarrow a$.
2. The minimum DAC associated with a variable X_i, $min_v \{dac_{iv}\}$, is a lower bound of the number of inconsistencies of any total assignment including X_i.
3. The sum of minimum DAC associated with the i last variables, $\sum_{j=i}^{n} min_v\{dac_{jv}\}$, for an arbitrary i, is a lower bound of the number of inconsistencies of any assignment including the last i variables.

Proof:

The proof of the first two statements is obvious. The third statement is true because each violated constraint can only contribute once to the expression because: (*i*) the expression only considers one DAC for each variable and (*ii*) each constraint does only contribute to DAC of its first variable regarding the ordering.

```
Function ini_DAC ()
1       for i:=1 to n do for all a∈Di do dacia:=0 endfor endfor
2       for i:=n-1 to 1 do
3               for j:=n to i+1 do
4                       for all a∈Di do
5                               if(arc_inconsistent(Xi,a,Xj)) then dacia:=dacia+1 endif
6                       endfor
7               endfor
8       endfor
9       return(DAC)
endfunction
function arc_inconsistent(Xi,a,Xj) returns boolean
10      arc_incons:=true
11      for all b ∈Dj while (arc_incons) do
12              if(not inconsistent(Xi←a, Xj←b)) then arc_incons:=false endif
13      endfor
14      return(arc_incons)
endfunction
```

Figure 4.1: Algorithm for DAC initialization assuming lexicographical variable ordering.

Example 4.2:

The following table shows all DAC counts of the previous example. The last row gives the minimum DAC for each variable.

DAC	X_1	X_2	X_3	X_4
a	1	1	1	0
b	1	2	0	0
c	2	1	1	0
min	1	1	0	0

From DAC, we obtain lower bounds on total assignments. For example, from $dac_{1\,c}$=2 one knows that any total assignment including $X_1 \leftarrow c$ violates at least two constraints (observation 4.1.1). From the minimum DAC of X_1 one knows that the assignment of X_1 will necessarily violate one constraint (observation 4.1.2). From the sum of minimum DAC, one knows that any total assignment violates at least two constraints (observation 4.1.3).

If DAC are computed in a pre-processing step and PFC selects variables for instantiation in the same order used for DAC computation, DAC counts of future values can be included in the lower bound. Wallace proposed the following expression as an alternative to (2.1).

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} + \sum_{j \in \mathbf{F}} min_v\{dac_{jv}\} \qquad (4.1)$$

The three contributions can be added because they refer to inconsistencies produced by different constraints: *distance* refers to violated constraints among past variables, the sum of minimum IC refers to constraints between past and future variables, and the sum of minimum DAC refers to constraints among future variables. Therefore, no constraint is considered more than once. Moreover, if the current assignment has not been propagated yet, one may add its DAC count, dac_{ia}, to the lower bound because inconsistencies between the current and future variables are not included in IC (see line 11 of Figure 4.2).

Following a similar reasoning, DAC can also be used to increase the lower bounds of future values. Value b of future variable X_j has the following associated lower bound,

$$distance + ic_{jb} + dac_{jb} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{dac_{kv}\} \qquad (4.2)$$

In what follows, we denote $LB1(S)$ and $LB1(S, X_j, b)$ the lower bounds that PFC uses for dead-end detection and future value pruning at node S (*i.e.*: expressions (2.1) and (2.3)); and $LB2(S)$ and $LB2(S, X_j, b)$ the previous lower bounds which include DAC information.

Figure 4.2 presents the first PFC algorithm using DAC. It includes a new parameter, DAC, which records directed arc-inconsistency counts.

Before search starts, DAC must be computed as indicated in Figure 4.1. Branch and bound search proceeds in the same way that PFC, but subject to DAC variable ordering and using the new lower bounds *LB2* instead of *LB1* (lines 11, 22 and 26). In the following, we will denote this algorithm PFC-DAC.

Example 4.3:

With the CSP of the previous example, consider a search state defined by the assignment $\{X_1 \leftarrow b\}$. At this point, future variables are X_2, X_3 and X_4. The look-ahead of the assignment causes the following inconsistency counts,

IC	X_2	X_3	X_4
a	0	1	1
b	1	1	1
c	1	0	1
min	0	0	1

at this node, PFC computes the following lower bound *LB1*,

$$distance + \sum_{j \in \{2,3,4\}} min_v\{ic_{jv}\} = 0 + 1 = 1$$

on the other hand PFC+DAC adds DAC information (see previous example) and computes the following lower bound *LB2*,

$$distance + \sum_{j \in \{2,3,4\}} min_v\{ic_{jv}\} + \sum_{j \in \{2,3,4\}} min_v\{dac_{jv}\} = 0 + 1 + 1 = 2$$

which detects one more constraint violation than PFC.

4.3 Theoretical Results on DAC Usage

PFC-DAC was presented as an improvement to PFC whose importance was only substantiated on a significant performance gain on some classes of overconstrained problems. Our following results theoretically complement the experimental evidence of the PFC-DAC advantage over plain PFC. We show that using DAC information is a major algorithmic enhancement to PFC which, at the cost of a polynomially expensive pre-process, can do exponentially better than PFC.

Lemma:

The algorithm presented for DAC computation requires a number of consistency checks in $O(n^2m^2)$.

```
procedure PFC-DAC ( P, F, dist, Assg, Dom, IC, DAC)
1       if (F = ∅) then
2               UB:= dist
3               Best_sol:= Assg
4       else
5               (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6               while (D_i≠∅) do
7                       a:= select_current_value(D_i)
8                       D_i:= D_i-{a}
9                       NAssg:= Assg ∪ {X_i←a}
10                      ndist:= dist + ic_ia

11                      if(ndist+ dac_ia + Σ_{j∈F-{i}} min_v{ic_jv}+ Σ_{j∈F-{i}} min_v{dac_jv} < UB) then

12                              (NDom,NIC):= look_ahead(ndist, X_i, a, F-{X_i},Dom-{D_i}, IC, DAC)
13                               if(not empty_domain(NDom)) then
14                                 PFC-DAC(P∪{X_i}, F-{X_i}, ndist, NAssg, NDom, NIC, DAC)
15                               endif
16                      endif
17              endwhile
18      endif
endprocedure
function look_ahead(ndist, X_i, a, F, Dom, IC, DAC)
19      stop:= false
20      for all D_j∈Dom while (not stop) do
21              for all b ∈D_j do

22                  if (ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv}+ Σ_{k∈F-{j}} min_c{dac_kc} ≥ UB) then

23                      D_j:= D_j-{b}
24                  else_if (inconsistent(X_i←a, X_j←b)) then
25                      ic_jb:=ic_jb+1

26                      if (ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv}+ Σ_{k∈F-{j}} min_v{dac_kv} ≥ UB) then

27                              D_j:= D_j-{b}
28                      endif
29                  endif
30              endfor
31              if(D_j= ∅) then stop:= true endif
32      endfor
33      return (Dom, IC)
endfunction
```

Figure 4.2: First version of PFC enhanced with DAC information based on [Wallace, 94] (PFC-DAC). Bold line numbers indicate differences with respect to PFC.

Theorem 4.1:

For any CSP and using the same variable and value selection, the number of consistency checks performed by PFC-DAC during search is no greater than the number of consistency checks performed by PFC.

Proof:

First, we show that all nodes visited by PFC-DAC are also visited by PFC. Observe that, because of the common variable and value ordering, the same search tree is traversed by both algorithms and in the same order. Hence, when visiting the same node, they have the same upper bound, the same distance and the same inconsistency counts for feasible values. Comparing PFC and PFC-DAC lower bounds ($LB1(S)$ and $LB2(S)$), it is clear that

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} \leq distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} + \sum_{j \in \mathbf{F}} min_v\{dac_{jv}\}$$

therefore, there cannot be a node such that PFC detects a dead-end, but PFC-DAC does not (recall observation 2.3, where we show that the empty domain condition for backtracking is also defined by the lower bound).

Second, we show that at nodes visited by both algorithms, PFC-DAC never performs more consistency checks than PFC. Note that consistency checks are only performed during the look-ahead. Comparing their pruning conditions ($LB1(S, X_j, b)$ and $LB2(S, X_j, b)$), it is easy to see that,

$$distance + ic_{jb} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} \leq distance + ic_{jb} + dac_{jb} +$$

$$+ \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}\} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{dac_{kv}\}$$

so, PFC-DAC always prunes values at higher tree levels than PFC. Once a value is pruned, its IC does not need to be updated, so it does not cost more consistency checks in the search below. Consequently, PFC-DAC look-ahead will never require more consistency checks than PFC look-ahead.

Theorem 4.2:

There are CSP instances for which PFC requires to perform a number of consistency checks exponentially larger than PFC-DAC.

Proof:

At every node, both PFC and PFC-DAC perform at least one consistency check, but never more than $O(n \cdot m)$ checks. Thus, visiting an exponential number of nodes requires an exponential number of checks, while visiting a polynomial number of nodes, requires a polynomial number of checks. We show that, there exists a CSP instance for which PFC visits an exponentially large number of nodes, so it requires an exponentially large number of consistency checks. For the same instance PFC-DAC only visits a polynomially large number of nodes. Then, PFC-DAC requires a polynomially large number of consistency checks (from the lemma, we know that DAC computation is also done with a polynomial number of checks). For the sake of clarity, but without loss of generality, we assume that both algorithms use a lexicographical ordering.

Consider a totally connected problem such that each of its constraints restricts all possible pairs of values (*i.e:* $R_{ij} = \varnothing \ \forall i,j=1,\ldots, n$). For this particular problem, all leaves have a distance of $n(n\text{-}1)/2$. Thus, after visiting the first leaf, both algorithms set the upper bound to $n(n\text{-}1)/2$.

Regarding PFC, consider an arbitrary node S in level i. It is easy to see that its current distance is $i(i\text{-}1)/2$. Besides, all future values are inconsistent with all past variables, so their IC takes value i. Then, the lower bound $LB1(S)$ is,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} = i(i\text{-}1)/2 + i(n\text{-}i)$$

It is easy to see that this lower bound is always lower than the upper bound, except for the tree level before the last one (namely, when i takes value n-1). Thus, PFC visits every tree node except the leaves (it only visits the first leaf). It is clear that this is an exponentially large number of nodes $m^{n-1}+1$. Figure 4.3 gives a visual idea of the PFC traversal on the totally constrained problem with 3 variables and 3 values per variable (solid lines indicate the traversed part of the tree).

Regarding PFC-DAC, for this particular problem every value is arc-inconsistent with every variable. Then, dac_{jb} takes value $(n\text{-}j)$ for all j and b. Consequently, an arbitrary node at level i has the following DAC contribution,

$$\sum_{j \in \mathbf{F}} min_v\{dac_{jv}\} = \sum_{j \in \mathbf{F}} (n\text{-}j) = (n\text{-}i\text{-}1)\ (n\text{-}i)/2$$

Therefore, the lower bound $LB2(S)$ is,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}\} + \sum_{j \in \mathbf{F}} min_v\{dac_{jv}\} = i(i\text{-}1)/2 + i(n\text{-}i) + (n\text{-}i\text{-}1)\ (n\text{-}i)/2$$

which is equal to the upper bound because

$$i(i\text{-}1)/2 + i(n\text{-}i) + (n\text{-}i\text{-}1)(n\text{-}i)/2 = n(n\text{-}1)/2$$

So, once the initial upper bound is set, the dead-end condition is true at every node. Then, PFC-DAC moves to the first leaf, sets the initial upper bound and backtracks all the way back to the root. At each node on the way back, it tries the remaining values, but the dead-end condition always holds. Consequently, PFC-DAC visits exactly $n \cdot m$ nodes which is obviously a polynomially large number. Figure 4.4 gives a visual idea of the traversal for the totally constrained problem of 3 variables and 3 values per variable.

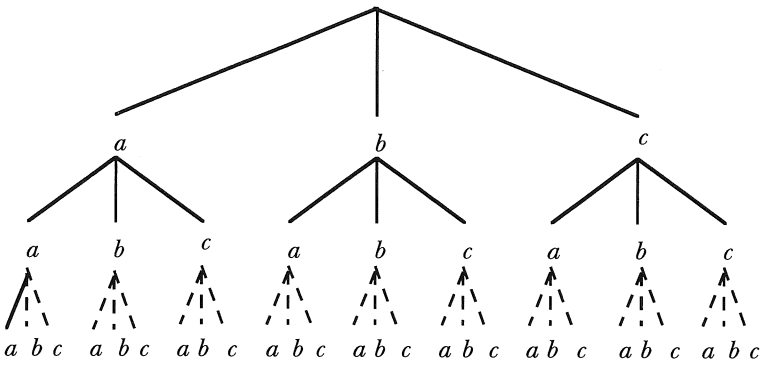


Figure 4.3: Tree traversal of PFC on a totally constrained CSP.

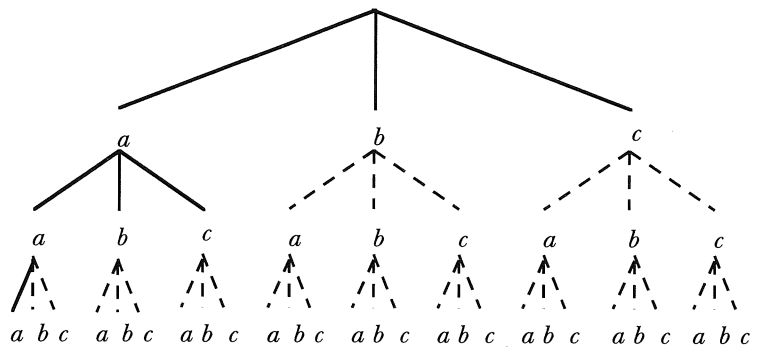


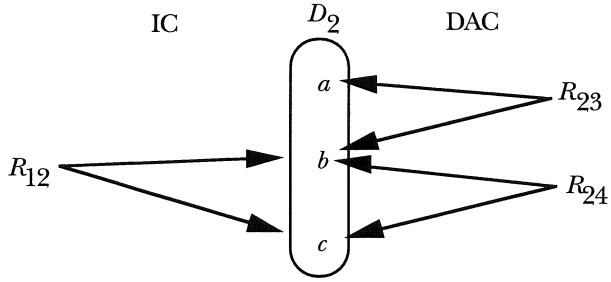
Figure 4.4: Tree traversal of PFC-DAC on a totally constrained CSP.

4.4 Combining DAC with IC[1]

Our first improvement to PFC-DAC makes a more profitable use of DAC counts. It is based upon the observation that DAC and IC counts of future values play a complementary role: IC count backward inconsistencies to past variables, subject to their current assignment, while DAC count forward inconsistencies to posterior future variables, independently of what value is assigned to them. Given that ic_{jb} and dac_{jb} have contributions from different sets of constraints, they can be added for each value b, obtaining the number of constraints that will necessarily be violated if value b is assigned to variable X_j keeping the current assignment

Example 4.4:

Consider the CSP and the search state of the previous example (*i.e.*: $\{X_1 \leftarrow b\}$). There are two different types of detected inconsistencies on a future variable. IC indicate inconsistencies of the considered variable with past variables, while DAC indicate inconsistencies of the considered variable with future variables posterior to it in the ordering. The following picture indicates constraint violations associated to X_2,



The arrows on the left represent inconsistencies detected by IC. The arrows on the right represent inconsistencies detected by DAC. For instance, value b violates one constraint (R_{12}) because of the past assignment $\{X_1 \leftarrow b\}$ and two more because of directional arc-inconsistencies (R_{23} and R_{24}). Consequently, as far the current assignment does not change, value b will necessarily violate three constraints (R_{12}, R_{23} and R_{24}). Inconsistencies detected in that way can also be added over variables because they refer to different constraints.

---

[1]The algorithm that includes the improvement described in this Section was presented in [Larrosa and Meseguer, 96b]. In that paper, it was denoted P-EFC3+DAC2.

Observation 4.2:

Consider an arbitrary node where X_j is a future variable. The following statements are true:

1. Let b be a feasible value of X_j. Then, $ic_{jb}+dac_{jb}$ is a lower bound of the number of inconsistencies caused by $X_j \leftarrow b$, if the current partial assignment is extended into a total one that includes $X_j \leftarrow b$.
2. $min_v\{ic_{jv} + dac_{jv}\}$ is a lower bound of the number of inconsistencies associated with the assignment of X_j if the current partial assignment is extended into a total one, regardless of the values assigned.
3. The expression $\sum_{j \in \mathbf{F}} min_v\{ic_{jv}+dac_{jv}\}$ is a lower bound of the number of inconsistencies caused by future variables if the current partial assignment is extended into a total one, regardless the values assigned.

Proof:

For the three statements, one can see that each constraint can only contribute once to the expression. Besides, each contribution reflects a detected inconsistency (IC capture inconsistencies with past variables, while DAC capture inconsistencies with posterior future variables).

Under these considerations, more powerful lower bounds can be computed. The following expression is a better lower bound associated with an arbitrary node,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}+dac_{jv}\} \tag{4.3}$$

In the following we will denote this expression $LB3(S)$, where S is the considered node. As with (4.1), if the lower bound is used before the propagation, one can add the current assignment DAC, dac_{ia}, because its information has not yet been propagated to IC. In addition, the following expression can be used during the look-ahead to test the feasibility of a value b in a variable X_j,

$$distance + ic_{jb} + dac_{jb} + \sum_{k \in \mathbf{F}\text{-}j} min_v\{ic_{kv}+dac_{kv}\} \tag{4.4}$$

We will refer to this expression as $LB3(S, X_j, b)$.

The use of these new lower bounds is an algorithmic improvement with respect PFC-DAC, as we indicate in the following observation.

Observation 4.3:

1. It is easy to see that $LB2(S) \leq LB3(S)$. Therefore, using $LB3(S)$ branch and bound never visits more nodes than using $LB2(S)$.
2. Comparing future value lower bounds, it is easy to see that $LB2(S, X_j, b) \leq LB3(S, X_j, b)$. Therefore, using $LB3(S, X_j, b)$ values are always pruned at higher nodes than using $LB2(S, X_j, b)$.
3. From (1) and (2), we know that using lower bounds $LB3(S)$ and $LB3(S, X_j, b)$ never causes search to visit more nodes and never requires more consistency checks. Thus, the algorithm that uses $LB3(S)$ and $LB3(S, X_j, b)$ instead of $LB2(S)$ and $LB2(S, X_j, b)$ is always better in terms of both visited nodes and consistency checks.

Bringing this improvement into practice requires the replacement of lines 11, 22 and 26 in Figure 4.2 by:

11 **if** $(ndist + dac_{ia} + \sum_{j \in F - \{i\}} min_v\{dac_{jv} + ic_{jv}\} < UB)$ **then**

22 **if** $(ndist + ic_{jb} + dac_{jb} + \sum_{k \in F - \{j\}} min_v\{ic_{kv} + dac_{kv}\} \geq UB)$ **then**

26 **if** $(ndist + ic_{jb} + dac_{jb} + \sum_{k \in F - \{j\}} min_v\{ic_{kv} + dac_{kv}\} \geq UB)$ **then**

Example 4.5:

Consider the CSP and the search state of the previous example. Adding DAC to the current future variable IC, we have

IC+DAC	X_2	X_3	X_4
a	1	2	1
b	3	1	1
c	2	1	1
min	1	1	1

at this point, with $LB3(S)$ the following lower bound is obtained,

$$distance + \sum_{j \in \{2,3,4\}} min_v\{ic_{jv} + dac_{jv}\} = 0 + 3 = 3$$

which detects one more violation than PFC+DAC and two more than PFC.

4.5 Saving Consistency Checks Associated with DAC

Our second improvement avoids the succesive repetition of consistency checks corresponding to detected arc-inconsistencies. Consider that we know that value $a \in D_i$ is arc-inconsistent with variable X_j. This knowledge compiles, in a sense, m consistency checks because it means that

$$(a,b) \notin R_{ij} \quad \forall b \in D_j$$

The use of DAC requires a pre-processing in which some arc-inconsistencies are detected. If these arc-inconsistencies are individually recorded, they can be reused during search saving their corresponding consistency checks. We introduce an additional data structure, *GivesDac*, that records the individual contribution of each variables to DAC. Thus, if $GivesDac_{iaj}$ is *true* (assuming $i<j$), it means that X_j is arc-inconsistent with $X_i \leftarrow$ a and, as a consequence, dac_{ia} has a contribution from X_j. The information contained in this data structure is gathered during the pre-processing step at no additional cost.

If $GivesDac_{iaj}$ is *true*, each time variable X_i becomes current and value a is assigned to it, look-ahead propagates the assignment toward future variables. In particular, the current assignment is propagated toward X_j. However, we do not need to check the consistency of the current assignment against X_j because we already know the result (they are inconsistent). Therefore, a first approach would be to increment by one the IC of every feasible value of D_j without any constraint check. However, incrementing every IC of feasible values of X_j has *exactly the same effect* as incrementing by 1 the current *distance* and leaving IC unaltered. In practice, it is more efficient to add dac_{ia} to the current distance and avoid updating IC corresponding to variables which contributed to dac_{ia}. To do this, we only need to replace lines 10, 11 and 24 in Figure 4.2 by:

```
10    ndist:= dist + ic_ia + dac_ia
11    if(ndist+ Σ_{j∈F-{i}} min_v{ic_jv+dac_jv}<UB) then
24    else_if (not(GivesDac_iaj) and inconsistent(X_i←a, X_j←b)) then
```

Lines 10 and 11 have the effect of including dac_{ia} to the current distance, while line 24 avoids redundant updating of IC, whose information is already reflected in the current distance.

Observation 4.4:

The algorithm that exploits *GivesDac* to avoid the repetition of known consistency checks visits exactly the same number of nodes because it

uses the same lower bounds. However, it never performs more consistency checks.

Example 4.6:

Consider the CSP and the search state of the running example. We have seen that $dac_{1\,b}$=1 because $X_1 \leftarrow b$ is arc-inconsistent to X_4. Therefore, $GivesDac_{1\,b4}$=*true*. If the assignment has been propagated without updating X_4 IC, we have the following situation,

IC	X_2	X_3	X_4
a	0	1	0
b	1	1	0
c	1	0	0

However, the lower bound remains unaltered if $dac_{1\,b}$ is added to the current distance. Thus, the lower bound is,

$$distance + \sum_{j \in \{3,4\}} min_v\{ic_{jv}+dac_{jv}\} = 1 + 2 = 3$$

which produces the detection of the same number of inconsistencies.

4.6 Graph-based DAC

Originally, [Wallace, 94] discarded the use of full arc-inconsistency counts (*i.e.* the arc-inconsistency count associated to value a of variable X_i is the number of variables that are arc-inconsistent with $X_i \leftarrow a$, with no ordering restriction) because their sum could record the same inconsistency twice (see Example 4.1), so they could not be used to compute lower bounds. Instead, directional arc-inconsistency counts were used because they do not present this problem. Following the work of [Dechter and Pearl, 88] on directed arc-consistency, a static variable ordering was required. However, the only purpose of establishing an ordering among variables and computing DAC under this ordering is to establish a direction for each individual constraint. Hence, each constraint can only contribute to DAC with one out of its two variables. Regarding lower bound computation, this restriction is arbitrary. Each constraint can only contribute to DAC in one of its two possible directions (otherwise DAC could not be safely added), but the selected direction must not be induced by a variable ordering.

With this idea in mind, we associate a *directed* graph, G, with a CSP in the following way: each problem variable corresponds to a vertex and each constraint, R_{ij}, defines one edge connecting its corresponding nodes.

The direction of the edge can be either (i,j) or (j,i) and will indicate the direction in which arc-inconsistencies are checked for DAC inclusion ((j,i) means that R_{ij} will contribute to DAC of X_i). Therefore, such a graph is not uniquely defined upon a given CSP. Its set of edges will be denoted by $EDGES(G)$. Given an arbitrary variable X_i, we will denote by $PREC(i,G)$ (respectively $SUCC(i,G)$) the set of variables X_j such that (j,i) (respectively (i,j)) is an edge of G. We define $DIRECTION(i,j,G)$ to be 1 if $(i,j) \in EDGES(G)$, -1 if $(j,i) \in EDGES(G)$, and 0 if there is no constraint between X_i and X_j.

Given any directed graph G of a CSP, one can define directed arc-inconsistency counts based on this graph, $dac_{ia}(G)$, in the following way.

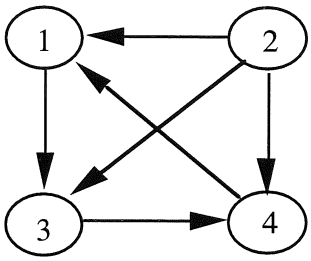
Definition 4.2:

Given a CSP and an arbitrary directed graph G, the *directed arc-inconsistency count* associated with value $a \in D_i$, $dac_{ia}(G)$, is the number of variables in $PREC(i,G)$ which are arc-inconsistent with $X_i \leftarrow a$.

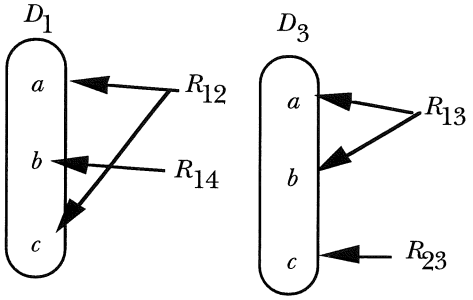
With this definition, DAC depend on the selected graph. In the following, we will omit the graph reference when it is clear by the context.

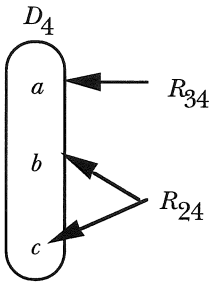
Example 4.7:

Consider the CSP of the previous examples. If we associate the following directed graph to it,



we have the following individual contributions to DAC counts.





Observe that each constraint only contributes to the variable indicated by the edge direction of G. For this particular G, X_2 does not receive any DAC contribution (*i.e:* its DAC counts are zero) because it does not have any predecessor.

Because of the direction given to each constraint, DAC can be added over variables without any constraint duplication. For instance, any solution including $X_1 \leftarrow a$ will violate one constraint (R_{12}); any solution including $X_3 \leftarrow a$ will violate one constraint (R_{13}); any solution including $\{X_1 \leftarrow a, X_3 \leftarrow a\}$ will violate two constraints (R_{12} and R_{13}). The following table shows DAC counts subject to graph G.

DAC(G)	X_1	X_2	X_3	X_4
a	1	0	1	1
b	1	0	1	1
c	1	0	1	1

The sum of minimum DAC gives an initial lower bound of 3 inconsistencies, which is one more inconsistency than when DAC were computed under lexicographical order.

In Figure 4.5 we show the algorithm for computing DAC subject to a directed graph. After setting every DAC to 0, it iterates on G edges and adds individual contributions to DAC subject to the edge direction. Observe that the previous version of *ini_DAC* (Figure 4.1) is just a particular case where G is induced by a variable ordering.

The interest of graph-based DAC is that they can also be used to compute lower bounds on the number of inconsistencies.

Observation 4.5:

Given a CSP, a directed graph G and its associated DAC, the following expression is a lower bound of the number of constraints that any total assignment will violate

$$\sum_{j=1}^{n} min_v\{dac_{jv}(G)\}$$

Proof:

As with lower bounds previously proposed, its correctness is based in the fact that each constraint can only contribute once to the expression. In this case, the reason is that graph G is directed with one edge per constraint, so each individual constraint has an associated direction under which arc-consistencies are computed.

When graph-based DAC are combined with IC, the situation becomes more involved. We cannot use the following lower bound during search,

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}+dac_{jv}(G)\}$$

because *distance*, DAC and IC can duplicate information. The problem of duplicated information arises when the current variable meets forward edges in G with future variables. Let X_j and X_i be two variables such that $(i,j) \in EDGES(G)$ and $GivesDac_{jbi}$ is $true$. In that situation, dac_{jb} has a contribution from constraint R_{ij}. We cannot select for instantiation X_i before X_j because independently of what value is assigned to X_i, its propagation will produce an increment of 1 in ic_{jb}. But the same inconsistency is already recorded in dac_{jb}, so adding $ic_{jb}+dac_{jb}$ would count the same inconsistency twice.

The solution to this problem is to compute DAC at each node subject to the subgraph, $G^{\mathbf{F}}$, associated to future variables only.

$$distance + \sum_{j \in \mathbf{F}} min_v\{ic_{jv}+dac_{jv}(G^{\mathbf{F}})\}$$

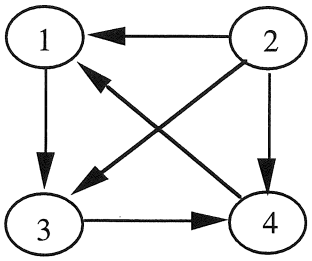
Then, DAC contributions from past variables are discarded. This situation is illustrated in the following example.

```
procedure ini_DAC (G)
1       for i:=1 to n do for all a∈D_i do dac_ia:=0 endfor endfor
2       for all (j,i)∈EDGES(G) do
3                       for all a∈D_i do
4                               if(arc_inconsistent(X_i,a,X_j)) then dac_ia:=dac_ia+1 endif
5                       endfor
6       endfor
7       return(DAC)
endprocedure
function arc_inconsistent(X_i,a,X_j) returns boolean
8       arc_incons:=true
9       for all b ∈D_j while (arc_incons) do
10              if(not inconsistent(X_i←a, X_j←b)) then arc_incons:=false endif
11      endfor
12      return(arc_incons)
endfunction
```

Figure 4.5: DAC initialization subject to a directed graph.

Example 4.8:

Consider our running example with its DAC computed subject to the directed graph G previously defined.

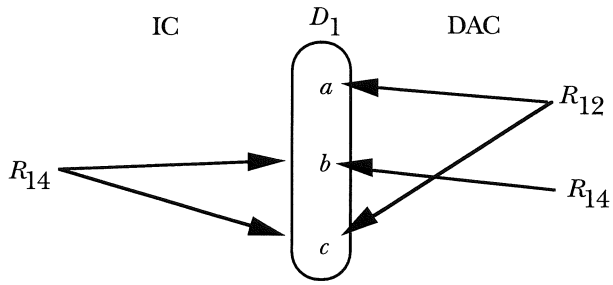


DAC(G)	X_1	X_2	X_3	X_4
a	1	0	1	1
b	1	0	1	1
c	1	0	1	1

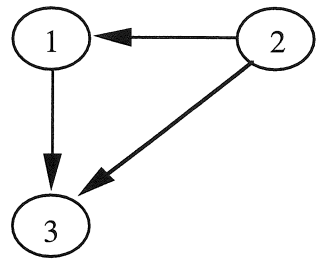
Observe that now it is senseless to select variables in lexicographical order because DAC are not computed subject to it. Let $X_4 \leftarrow a$ be the first assignment attempted. After the propagation, the following IC are computed,

IC	X_1	X_2	X_3
a	0	0	1
b	1	1	1
c	1	1	1

We cannot combine DAC based on G with these IC because X_4 has a forward edge in G (*i.e:* $(4,1) \in EDGES(G)$). In this particular case, ic_{1b} and dac_{1b} refer to the same constraint violation (R_{14}). It is illustrated in the following picture which shows detected inconsistencies in D_1,



To circumvent that problem, one should use the subgraph, $G^{\mathbf{F}}$, which only includes future variables. Therefore, it does not include any DAC contribution from X_4.



It produces the following arc-inconsistency counts,

DAC($G^{\mathbf{F}}$)	X_1	X_2	X_3
a	1	0	1
b	0	0	1
c	1	0	1

Which do not duplicate any inconsistency so they can be safely added,

IC+DAC	X_1	X_2	X_3
a	1	0	2
b	1	1	2
c	2	1	2
min	1	0	2

giving a lower bound of 3 inconsistencies.

The problem now is that DAC change during search. Each time search moves to a successor, DAC must be updated removing those arc-inconsistency contributions caused by the current variable. Interestingly, it is not necessary to compute DAC from scratch at each node. We can use the *GivesDac* structure to detect the situation and decrement redundant DAC. If X_i is the current variable, we must decrement all dac_{jb} such that there is a forward edge with their variable (*i.e:* $DIRECTION(i,j,G)$=1) and X_i contributed to their DAC (*i.e:* $GivesDac_{jbi}$ is *true*) because the propagation will increment their IC producing an information duplication. However, incrementing ic_{jb} and decrementing dac_{jb} is irrelevant with respect branch and bound. Thus, in practice it is more efficient to leave DAC unaltered and prevent the updating of those IC of future variables whose DAC has a X_i contribution. To do this, we only need to replace line 24 of Figure 4.2 by:

```
24    else_if ((DIRECTION(i,j,G)=-1 and not(GivesDac_iaj)) or
                  (DIRECTION(i,j,G)=1 and not(GivesDac_jbi))) and
                  inconsistent(X_i←a, X_j←b)) then
```

The first disjunction is necessary for the improvement presented in Section 4.5. The second disjunction is necessary to prevent the graph-based duplication discussed above. In this way it is guaranteed that if we increment ic_{jb}, it does not have any contribution already included in dac_{jb}. Consequently, IC and DAC of future values can be safely added to form a lower bound of inconsistencies for that value.

Observation 4.6:

> An additional advantage of this new improvement is that it unrelates DAC with the variable ordering heuristic. Thus, with this approach, branch and bound *can be combined with dynamic variable ordering heuristics*. This is beneficial because, in general, dynamic variable orderings are more effective than static variable orderings.

The interest of graph-based DAC is that they generalize the previous concept of DAC based on a variable ordering. In fact, DAC based on a variable ordering are equivalent to DAC based on a directed graph where edges are induced by the ordering. Previous work on DAC usage computed DAC subject to a variable ordering because, if DAC are computed under that ordering and branch and bound follows the same ordering for variable instantiation, the current variable never meets forward edges, so the problem of redundant information when combining DAC and IC does not occur. However, we have shown that this problem can be solved by means of the *GivesDac* structure, with no computational overhead at all.

In the classical DAC usage one needs to decide a variable ordering before computing DAC. With this new approach, one needs to decide the direction of *each constraint*. Thus, it allows for more freedom in the way arc-inconsistencies contribute to the lower bound (there is a factorial number of possible total orderings among variables and an exponential number of possible directed graphs). An additional advantage of graph-based DAC is that they do not require the use of static variable ordering, which was believed to be necessary when using DAC [Wallace, 94; Larrosa and Meseguer, 96].

The algorithm that includes the improvements introduced in Sections 4.4-4.6, denoted PFC-GDAC (because of its graph-based DAC), is presented in Figure 4.6. It assumes a directed graph *G*, under which DAC have been computed during a pre-process (as indicated in Figure 4.5).

```
procedure PFC-GDAC ( P, F, dist, Assg, Dom, IC, DAC, G)
1       if (F = ∅) then
2               UB:= dist
3               Best_sol:= Assg
4       else
5               (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6               while (D_i≠∅) do
7                       a:= select_current_value(D_i)
8                       D_i:= D_i-{a}
9                       NAssg:= Assg ∪ {X_i←a}
10                      ndist:= dist + ic_ia + dac_ia
11                      if (ndist+ Σ_{j∈F-{i}} min_v{ic_jv+dac_jv} < UB) then
12                        (NDom,NIC):= look_ahead(ndist, X_i, a, F-{X_i},Dom-{D_i},IC, DAC, G)
13                        if(not empty_domain(NDom)) then
14                              PFC-GDAC(P∪{X_i}, F-{X_i},ndist, NAssg, NDom, NIC, DAC, G)
15                        endif
16                      endif
17              endwhile
18      endif
endprocedure
function look_ahead(ndist, X_i, a, F, Dom,IC, DAC, G)
19      stop:= false
20      for all D_j∈Dom while(not stop)do
21              for all b ∈D_j do
22                      if (ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv+dac_kv} ≥ UB) then
23                              D_j:= D_j-{b}
24                      else_if ((DIRECTION(i,j,G)=-1 and not(GivesDac_iaj)) or
                                        (DIRECTION(i,j,G)=1 and not(GivesDac_jbi))) and
                                                inconsistent(X_i←a, X_j←b)) then
25                              ic_jb:=ic_jb+1
26                              if (ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv+dac_kv} ≥ UB) then
27                                      D_j:= D_j-{b}
28                              endif
29                      endif
30              endfor
31              if(D_j= ∅) then stop:= true endif
32      endfor
33      return (Dom,IC)
endfunction
```

Figure 4.6: PFC-GDAC. Bold line numbers indicate differences with respect PFC-DAC.

4.7 Reversible DAC

In the previous Section, we showed that DACs could be computed subject to a directed graph, instead of a variable ordering. It means that each constraint has an associated direction under which its arc-inconsistencies are exploited. In this Section we go one step further. We consider the dynamic rearrangement of the graph during search, targetting a lower bound maximization subject to current IC.

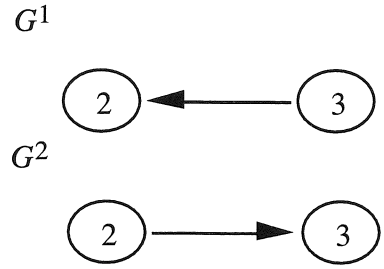
Using graph-based DAC, as presented in the previous Section, requires the choice of the graph before search starts. This decision is important since it determines those arc-inconsistencies that contribute to the lower bound during search. Obviously, one would like to find the *optimal* graph (*i.e.*: find a direction for each constraint such that their directional arc-inconsistencies combined with IC cause the maximal contribution to the lower bound). However, IC change during search, while the graph remains static. Therefore, a good graph for some nodes can be bad for others. Consequently, graph optimality is a dynamic concept which depends on the visited node.

Example 4.9:

Consider a CSP having three variables, two values per variable and the following constraints: $R_{12}=\{(a,b), (b,a), (b,b)\}$, $R_{13}=\{(a,a), (a,b), (b,b)\}$ and $R_{23}=\{(a,a)\}$. Let S^a be the node defined by the partial assignment $\{X_1 \leftarrow a\}$. It is easy to see that the propagation produces the following inconsistency counts,

IC(S^a)	X_2	X_3
a	1	0
b	0	0

Regarding DAC contribution, there is only one constraint between future variables (R_{23}). Hence, there are two possible directed subgraphs for the current future variables (one for each possible direction of the future constraint). Let G^1 denote the graph with edge (3,2) and G^2 denote the graph with edge (2,3).



Each one produces different DAC counts,

DAC(G^1)	X_2	X_3
a	0	0
b	1	0

DAC(G^2)	X_2	X_3
a	0	0
b	0	1

At this search state, it is more suitable to use DAC based on G^1 because they contribute precisely to the value having the minimum IC in X_2. Thus, adding IC+DAC increases the lower bound in one unit. On the other hand, if we use DAC based on G^2, they do not give any advantage over plain IC because their only contribution is not enough to increase the minimum IC+DAC of X_3.

IC(S^a)+DAC(G^1)	X_2	X_3
a	1	0
b	1	0

IC(S^a)+DAC(G^2)	X_2	X_3
a	1	0
b	0	1

Consider now a node S^b defined by the partial assignment $\{X_1 \leftarrow b\}$. S^b has the following inconsistency counts,

IC(S^b)	X_2	X_3
a	0	1
b	0	0

In this node, we have the opposite situation. Using DAC based on G^1 is useless because its only contribution cannot increase the minimum IC+DAC of X_2. On the other hand, using DAC based on G^2, there is a contribution to value $b \in X_3$, which is enough to increase the lower bound in one unit.

IC(S^b)+DAC(G^1)	X_2	X_3
a	0	1
b	1	0

IC(S^b)+DAC(G^2)	X_2	X_3
a	0	1
b	0	1

The only reason for using the same graph throughout all search is that its DAC can be used without extra recomputing overhead during search. However, given that no static graph is globally optimum, one may think of changing the graph during search. That means giving to each constraint the direction in which its arc-inconsistencies are more useful, subject to the current IC. Obviously, changing the graph requires some additional effort at each search node. There is a trade-off between the cost of finding a good graph for each node and the search savings that it may cause.

At each node, one might like to find an optimal directed graph subject to the current set of future variables and the current IC (*i.e.* decide the direction to each individual constraint such that combining its DAC with the current IC provides the *highest* lower bound). However, the number of possible graphs at a given node is exponentially large, and finding the optimal graph at each search state does not seem to be easily done in an efficient manner. To circumvent that drawback, we propose a simpler approach: we will not search for optimality, but only for sub-optimality. In the following we present a greedy heuristic which searches for a *good* graph at each node.

Our approach is based on the concept of *reversible* DAC (RDAC). The idea behind it is to reverse the edges of the graph dynamically during search with the objective of exploiting arc-inconsistencies in the direction in which they produce a higher contribution to the lower bound. The first requirement to bring this idea into practice in an efficient way is to detect all arc-inconsistencies prior to search. Recall that previous algorithms only search for arc-inconsistencies in the direction induced by the graph edges. In the following we assume that *GivesDac* structure has been initialized recording arc-inconsistencies in both directions (even those not contributing to any DAC count). Note that this inicialization requires

some additional computation at the pre-processing step. Hence, before search starts we have,

$$(GivesDac_{kcj} = true \Leftrightarrow c \in D_k \text{ is arc-inconsistent with } X_j) \qquad \forall k,c,j$$

With this data structure, reversing an edge and updating DAC counts accordingly can be done efficiently during search. Let X_j and X_k be two constrained future variables such that $(j,k) \in EDGES(G)$. X_k may then have arc-inconsistency contributions from X_j in its DAC. If the edge (j,k) is reversed, we only need to decrease all X_k DAC such that $GivesDac_{kcj}$ is *true* by one, and increase all X_j DAC such that $GivesDac_{jbk}$ is *true* by one. Figure 4.7 shows how it can be done. Procedure *reverse*(j,k,G) removes the edge (j,k) from G, adds the new edge (k,j) and updates DAC accordingly.

```
procedure reverse(j, k, DAC, G)
1       EDGES(G):= (EDGES(G)∪(k,j))-(j,k)
2       for all c ∈D_k do if (GivesDac_kcj) then dac_kc := dac_kc -1 endif endfor
3       for all b ∈D_j do if (GivesDac_jbk) then dac_jb := dac_jb +1 endif endfor
4       return(DAC,G)
endprocedure
```

Figure 4.7: Reversing an edge and updating DAC, as needed in PFC-RDAC.

In our work we follow a simple, computationally efficient heuristic. Each node inherits G (and, consequently, its associated DAC) from its parent. Then, each constraint between future variables is considered: If its reversal improves the current lower bound, then it is reversed; otherwise it is not. The process iterates until no local change can possibly improve the current lower bound. Observe that this is a typical hill climbing schema for local optimization. It makes local changes (constraint reversals) as far as they produce an immediate gain. It stops when it gets trapped in a local optimum. In general, local optima do not imply global optimality. Thus, our approach does not guarantee the obtention of the best graph for the current node.

Regarding a single edge reversal, there is a necessary condition for a reversal to increase the lower bound that can be easily tested.

Observation 4.7:

Let R_{jk} be a constraint between future variables such that $(j,k) \in EDGES(G)$. Let $b \in D_j$ and $c \in D_k$ denote those values having the minimum IC+DAC at the current node (*i.e:* $b = arg\_min_v\{dac_{jv}+ic_{jv}\}$ and $c = arg\_min_v\{dac_{kv}+ic_{kv}\}$). Then,

1. Reversing edge (j,k) does not produce a lower bound gain if in its current direction it contributes to the DAC of c. Or, what is equivalent,

$$GivesDac_{kcj} = true$$

2. Reversing edge (j,k) does not produce a lower bound gain if, in the opposite direction it does not contribute to the DAC of b. Or, what is equivalent,

$$GivesDac_{jbk} = false$$

Consequently, the following expression is a necessary condition for a reversal producing a lower bound increment that can be efficiently tested,

$$GivesDac_{kcj} = false \textbf{ and } GivesDac_{jbk} = true$$

Example 4.10:

Consider the previous example CSP. Let $\{X_1 \leftarrow a\}$ be the current assignment and

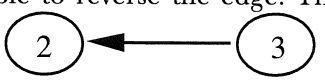
the current subgraph subject to future variables. In this situation, we have the following DAC+IC.

IC+DAC	X_2	X_3
a	1	0
b	0	1
min	0	0
arg_min	b	a

Which produces a zero lower bound. Values having the minimum IC+DAC in X_2 and X_3 are b and a, respectively. Thus, the necessary condition for reversing the edge (3,2) is,

$$GivesDac_{3a2} = false \textbf{ and } GivesDac_{2b3} = true$$

or, in words, that the current edge direction does not contribute to the minimum IC+DAC of X_3, and if it is reversed it contributes to the minimum IC+DAC of X_2. It is easy to see that this condition holds so it is reasonable to reverse the edge. The resulting graph is,



which produces the following DAC+IC,

IC+DAC	X_2	X_3
a	1	0
b	1	0
min	1	0
arg_min	a,b	a,b

This new situation allows for the detection of 1 inconsistency. Observe that now there are two values having the minimum IC+DAC at each variable. If we test again the necessary condition for reversing the edge (3,2), depending on what values are selected, the test will fail or succeed (for instance, if the test is performed using $a \in D_2$ and $b \in D_3$ it succeeds). However, reversing the edge returns to the previous situation which decreases the lower bound. This example proofs that the test is only a necessary condition.

Figure 4.8 shows *greedy_opt*, the procedure in charge of optimizing the current directed graph G, subject to the current IC. At each iteration, the current lower bound is recorded (line 3). It then iterates on the set of future constraints. For each constraint, we test whether or not reversing it can produce a lower bound increment following the idea of Observation 4.7 (line 9). If the test succeeds, it is reversed (line 10). If it fails, its direction is not changed. After each reversal, the new lower bound is computed. Only when the reversal is not counter-productive (line 11) it is maintained. If a whole iteration over constraints does not produce any gain, the procedure stops. It returns the new graph and its associated DAC.

The algorithm that reverses edges searching for a good graph at each visited node is denoted PFC-RDAC. It only differs from PFC-GDAC in that after propagating each assignment, procedure *greedy_opt* performs the greedy optimization of the lower bound. To do this, an additional line has to be added to PFC-GDAC.

```
13b        (NG, NDAC):= greedy_opt(G,DAC,F)
```

and line 14 has to be replaced by:

```
14         PFC-RDAC(P∪{X_i}, F-{X_i}, ndist, NAssg, NDom, NIC, NDAC, NG)
```

4.8 Maintaining DAC During Search

All previous algorithms have one feature in common: their DAC counts are based on arc-inconsistencies that are detected during a pre-process before search. However, forward checking algorithms prune future values that are found unfeasible. It may happen that value pruning causes new arc-inconsistencies which only hold at the current subproblem. Pre-computed DAC do not include these new arc-inconsistencies because they did not hold at the original problem. Thus, when branch and bound solves a subproblem is not using updated DAC with respect to this subproblem.

Our last improvement maintains DAC updated during search by including contributions of new arc-inconsistencies caused by pruned values. Updated DAC are always greater than or equal to pre-computed DAC. Hence, they lead to better lower bounds which increase value pruning. This new pruning can cause new arc-inconsistencies that are again reflected in higher DAC. Consequently, a cascade effect that will anticipate dead-end detection is expected.

```
function greedy_opt(G, DAC, F)
1       stop:= false
2       while(not stop) do
3               save_min:= Σ_{j∈F} min_v{dac_jv+ic_jv}
4               for all (j,k)∈EDGES(G) if (j∈F and k∈F) do
5                       min_j:= min_b{dac_jb+ic_jb}
6                       min_k:= min_c{dac_kc+ic_kc}
7                       val_min_j:= argmin_b{dac_jb+ic_jb}
8                       val_min_k:= argmin_c{dac_kc+ic_kc}
9                       if(not GivesDac_k val_min_k j and GivesDac_j val_min_j k) then
10                              (G,DAC) := reverse(j,k,G)
11                              if(min_v{dac_jv+ic_jv}+min_v{dac_kv+ic_kv}<min_j+min_k)then
12                                      (G,DAC) := reverse(k,j,G)
13                              endif
14                      endif
15              endfor
16              if(save_min = Σ_{j∈F} min_v{dac_jv+ic_jv}) then stop:= true endif
17      endwhile
18      return(G,DAC)
endprocedure
```

Figure 4.8: Greedy optimization of graph G, subject to the current IC.

We will refer to this approach as *maintaining directional arc-consistency* (MDAC) because of its parallelism with *maintaining arc-consistency* (MAC) in the total constraint satisfaction context. Observe that both algorithms follow the same idea: they detect new arc-inconsistencies caused by pruned values. However, in MAX-CSP arc-inconsistent values cannot be pruned. They are only recorded and added to DAC counts for lower bound computation. For this reason, it is enough to detect new arc-inconsistencies between future variables in the direction indicated by the corresponding graph edge.

To perform MDAC, any arc-consistency algorithm (along with the adequate data structures) can be used. The only requirement is that the effect of each value removal must be propagated toward future variables, increasing DAC accordingly. The algorithm that improves PFC-GDAC maintaining its DAC updated is denoted PFC-MDAC. It only requires the addition of a function call *propagate_del* after each value pruning.

Observation 4.8:

PFC-MDAC never causes the algorithm to visit more nodes because updating DAC always produces higher lower bounds. However, propagating each value removal causes an overhead during search which may or may not be cost-effective.

An AC-4 based implementation of *propagate_del* is given in Figure 4.9. It receives as input the support-based data structures that are assumed to be initialized before search. When value b of X_j is removed, the procedure iterates on the set of values to which the removal may affect (namely, future values such that there is a forward edge from connecting X_j and their variable, and such that b was giving support to them (line 1)). For each value, its support is decremented (line 2). If it becomes zero, it means that their last supporting value at X_j has been removed, so their DAC is incremented (line 3). The only difference between propagating a value deletion in total constraint satisfaction or in partial constraint satisfaction is that, in the latter case, those values that become arc-inconsistent have their DAC increased instead of being pruned.

The *LSupport* and *CSupport* data structures are useless when DAC are not maintained. However, one can observe that $CSupport_{iaj}=0 \Leftrightarrow GivesDac_{iaj}=true$. Therefore, the *GivesDac* structure is redundant with the MDAC-4 algorithm.

Maintaining DAC is easily combined with reversible DAC. The resulting algorithm, PFC-MRDAC, only requires the following two modifications:

1. RDAC requires the *GivesDac* structure to be updated with respect to both directions of each constraint in order to reverse DAC efficiently. Thus, in MRDAC, where *Csupport* plays the same role, it has to be updated in both directions, as well. To do this, one must modify the *propagate_del* procedure. Its pseudo-code appears in Figure 4.10. Its difference with Figure 4.9 is that *Csupport* is updated, regardless of the edge direction (line 2), but dac_{kc} is only incremented if new arc-inconsistencies occur in the appropriate direction (line 3).
2. The second modification is needed because RDAC improves the lower bound after each look-ahead and it may produce the unfeasibility of new values. In MRDAC, it is especially important to detect and propagate these new removals because it may produce new DAC contributions. To do this, we add a new function call after the *greedy_opt* procedure. This function, *prune_values*, is in charge of value deletion once the lower bound has been improved by the *greedy_opt* function. It appears in Figure 4.11. It can be seen as a simplified look-ahead with no IC updating. Observe that iteration continues until a fix point is reached. This has been found to be useful in practice, although gains are minor.

For the sake of completeness, PFC-MRDAC —the algorithm that includes all improvements presented in this chapter— is presented in Figure 4.12. It assumes graph G, DAC and AC-4 data structures to be properly initialized before search.

```
function propagate_del (X_j, b, F, G, CSpport, LSupport, DAC)
1       forall (X_k,c)∈LSupport_jb if (DIRECTION(j,k)=1 and X_k∈F) do
2                       CSupport_kcj := CSupport_kcj -1
3                       if (CSupport_kcj = 0) then dac_kc:=dac_kc +1 endif
4       endfor
5       return(CSupport, DAC)
endfunction
```

Figure 4.9: AC-4 based propagation of a value removal as needed in PFC-MDAC.

```
function propagate_del (X_j, b, F, G, CSpport, LSupport, DAC)
1       forall (X_k,c)∈LSupport_jb if (X_k∈F) do
2               CSupport_kcj := CSupport_kcj -1
3               if (CSupport_kcj =0 and DIRECTION(j,k)=1) then
4                       dac_kc:=dac_kc +1
5               endif
6       endfor
7       return(CSupport, DAC)
endfunction
```

Figure 4.10: AC4-based propagation of a value removal as needed in PFC-MRDAC.

```
function prune_values(ndist, F, Dom, G, CSupport, LSupport, DAC)
1       stop:= false
2       while (not stop) do
3               save_min:= Σ_{j∈F} min_v{dac_jv+ic_jv}
4               for all X_j∈F while (not stop) do
5                       for all b ∈D_j do
6                          if(ndist+ ic_jb+ dac_jb+ Σ_{k∈F-j} min_v{ic_kv+dac_kv}≥UB)then
7                              D_j:= D_j-{b}
8                              (CSupport,DAC):=
                                       propagate_del(X_j,b,F,G,CSupport,LSupport,DAC)
9                          endif
10                      endfor
11                      if (D_j = ∅) then stop:= true endif
12              endfor
13              if (save_min = Σ_{j∈F} min_v{dac_jv+ic_jv}) then stop:= true endif
14      endwhile
15      return(Dom, CSupport, DAC)
endfunction
```

Figure 4.11: Pruning values after a lower bound improvement, as required in PFC-MRDAC.

```
procedure PFC-MRDAC (P, F, dist, Assg, Dom, IC, DAC, G, CSupport, LSupport)
1 if (F = ∅) then
2       UB:= dist
3       Best_sol:= Assg
4 else
5       (X_i,D_i):= select_current_variable_and_domain(F, Dom)
6       while (D_i≠∅) do
7             a:= select_current_value(D_i)
8             D_i:= D_i-{a}
9             NAssg:= Assg ∪ {X_i←a}
10            ndist:= dist + ic_ia + dac_ia

11            if(ndist+ Σ_{j∈F-{i}} min_v{ic_jv+dac_jv}<UB) then
12                (NDom,NIC,NDAC,NewCSupport):=
                   look_ahead(ndist,X_i,a,F-{X_i},Dom-{D_i},IC,DAC,G, CSupport, LSupport)
13                if(not empty_domain(NDom)) then
13b                   (NG, NDAC):= greedy_opt(G,NDAC,F)
                      (NDom, NewCSupport, NDAC):=
13c                     prune_values(ndist,F, NDom, NG, NewCSupport, LSupport, NDAC)
14                    PFC-MRDAC(P∪{X_i},F-{X_i},ndist,NAssg,NDom,NIC,NDAC,NG,
                                                          NewCSupport,LSupport)
15                    endif
16            endif
17      endwhile
18 endif
endprocedure
function look_ahead(ndist, X_i, a, F, Dom, IC, DAC, G, CSupport, LSupport)
19      stop:= false
20      for all D_j∈Dom while(not stop)do
21            for all b ∈D_j do

22              if(ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv+dac_kv}≥UB)then
23                  D_j:= D_j-{b}
23b                 (CSupport,DAC):= prune_values(ndist,F,Dom,G,CSupport,LSupport,DAC)
24                elseif ((DIRECTION(i,j,G)=-1 and not(GivesDac_iaj)) or
                                  (DIRECTION(i,j,G)=1 and not(GivesDac_jbi))) and
                                        inconsistent(X_i←a, X_j←b)) then
25                  ic_jb:=ic_jb+1

26                  if(ndist+ ic_jb+ dac_jb+ Σ_{k∈F-{j}} min_v{ic_kv+dac_kv}≥UB)then
27                    D_j:= D_j-{b}
27b                   (CSupport,DAC):=prune_values(ndist,F,Dom,G,CSupport,LSupport,DAC)
28                    endif
29                endif
30            endfor
31            if(D_j= ∅) then stop:= true endif
32      endfor
33      return (Dom,IC, DAC, CSupport)
endfunction
```

Figure 4.12: PFC-MRDAC. Bold line numbers indicate differences w.r.t. PFC-GDAC.

4.9 Experimental Results

4.9.1 The MAX-CSP Complexity Peak[2]

In this Subsection, we examine the existence of a pattern in MAX-CSP average difficulty of problems. We show that the search effort of PFC enhanced with the use of directed arc consistency counts (DAC) presents and *easy-hard-easy* pattern when solving random binary CSP instances. Interestingly, if the algorithm does not use any local consistency information in its lower bound, the easy-hard-easy pattern does not occur and problems become increasingly hard. The *peak* in the search effort is related with a sudden change in the number of arc-inconsistencies of problems.

In recent years, increasing attention has been devoted to the phenomenon of phase transition appearing when solving different types of NP-complete problems, such as graph coloring, SAT and binary CSP [Cheeseman *et al.,* 91]. In these problems, the plot of the search effort shows an easy-hard-easy pattern when varying a certain order parameter. This pattern reflects a phase transition in the probability of problem solvability, which passes suddenly from 1 to 0 varying the order parameter. Typically, problems in the left-easy part are solvable and they have many solutions, so it is easy to find one. Problems in the right-easy part are unsolvable and it is also easy to find it out. The hard part, situated between the two easy parts and where the phase transition occurs, is composed of problems which are either solvable or unsolvable with an approximate proportion of 50% each. In this part, solvable problems have few solutions, so it is costly to find one, and unsolvable problems have many almost-solutions, so algorithms have to invest a lot of effort to finally find that no solution exists.

Regarding binary CSP [Prosser, 94], the complexity peak becomes apparent when varying constraint tightness. Experiments on the four parameter model for random problems show that if the number of variables (n), domain cardinality (m) and graph connectivity (p_1) is maintained and constraint tightness (p_2) is varied, the most difficult instances occur precisely at the point where problems suddenly change from solvable to unsolvable. Figure 4.13 illustrates this phenomenon using FC on the <10,10,45/45,p_2> class. For each problem class, the average number of consistency checks on samples of 50 instances is reported. We

[2]The discovery of this phenomenon was presented in [Larrosa and Meseguer, 96b] as a phase transition. Now, we prefer to call it a complexity peak. The reason is that we do not know whether the peak is related to a sudden change of some problem property or not.

also show the ratio of solvable problems which presents the behaviour previously described.

So far, phase transition research on CSP has been limited to the total constraint satisfaction case. In the following, we show that there is a complexity peak on MAX-CSP which becomes apparent if DAC are used to improve branch and bound lower bound. We experimented on the classes $<10,10,p_1,p_2>$ with the following values for p_1: 15/45, 25/45, 35/45 and 45/45. p_2 varies in steps of one hundredth within the range of oversconstrained instances (*i.e.* after the critical point where problems become unsolvable). For each parameter setting, we generated samples of 50 instances. At each class of problems, we compare PFC vs. PFC-DAC (as described in Figure 4.2). In this experiment we did not want variable and value ordering heuristics to distort the algorithms behaviour. For this reason, variables and values were selected in lexicographical order.

Figures 4.14 and 4.15 report the average number of consistency checks for these problem classes. Results in terms of CPU time and visited nodes are omitted because they present exactly the same behaviour. Figure 4.14 shows the results obtained with PFC. We clearly see that the search effort grows monotonically with tightness and this growth is exponential. Note that these results are on average: for a given problem instance, increasing the number of nogoods does not necessarily makes it harder to solve. In conclusion, no easy-hard-easy pattern is observed for these classes of problems when solved with PFC.

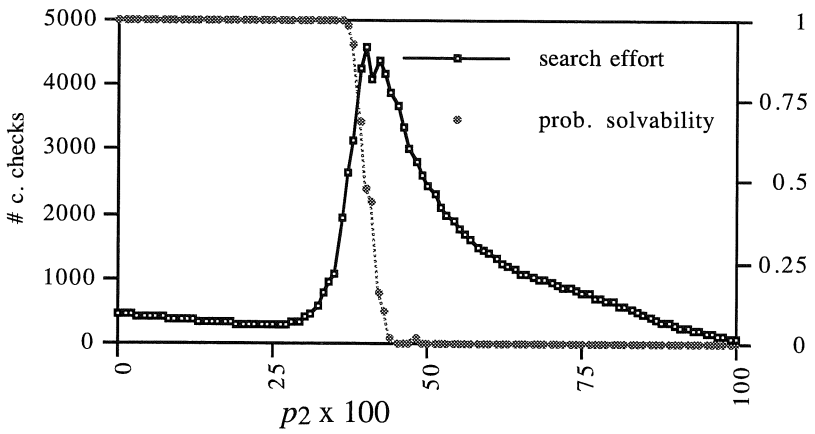


Figure 4.13: Phase transition and associated complexity peak on the $<10,10,1,p_2>$.

Figure 4.15 shows the results obtained with PFC-DAC. PFC-DAC improves PFC lower bound with arc-inconsistency information. This strategy has little or no effect for low values of p_2 because every DAC is likely to be zero. However, DAC efficiency increases for high values of p_2, when many constraints between future variables are arc-inconsistent. When tightness is increased, we observe a substantial decrease in the search effort required by PFC-DAC with respect to PFC (observe the difference in the consistency checks scale). More importantly, we observe a peak in the search effort after which the search cost falls drastically until it is almost zero for p_2=1. In addition, the hardest problems for PFC, located in the neighbourhood of p_2=1 are extremely easy to solve for PFC-DAC.

Given that MAX-CSP is an optimization problem, the easy-hard-easy pattern observed cannot be explained in terms of a phase transition in the probability of solution. Nevertheless, the observed peak depends on the evolution of lower bounds with p_2. For high values of p_2, there are many arc-inconsistencies which are reflected in DAC counts. There is a point after which the number of arc-inconsistencies starts growing very fast. This causes an abrupt increase of PFC-DAC lower bound, even at the highest tree levels. It has a drastic effect in the ability of PFC-DAC to anticipate pruning and produce empty domains. As a result, it causes a rapid decrease of the search effort required.

We can explain the behaviour of PFC-DAC in terms of upper and lower bounds. On the left side of the peak there is the region of low and medium tightness problems, for which the best solution violates none or a few constraints. On these problems branch and bound soon finds a low upper bound, such that low lower bounds are enough to prune, making problems easy to solve. On the right side of the peak there is the region of very high tightness problems, for which the best solution violates many constraints, that is, they present high upper bounds. For these problems DAC are also high, causing high lower bounds at shallow levels of the tree. As a result, the algorithm can prune efficiently and these problems are also relatively easy to solve. Around the peak there is the region of high tightness problems, for which the best solution has an important number of violated constraints (high upper bounds). However, the sum of minimum DAC is zero or very low, so branch and bound has to go deep in the tree to accumulate enough IC to perform pruning.

Figure 4.16 illustrates this situation on the $<10,10,1,p_2>$ class. It depicts the search effort needed by PFC-DAC and the evolution of bounds with p_2. Since bounds change during search we can only report some selected values. Regarding the upper bound, we plot for each class the average best solution, which is an under-estimation of the upper bound during search. Regarding the lower bound, we plot its value at the tree root (that is to say, the sum of minimum DAC), which is an under-estimation of the lower bound value during search. We can observe that these selected upper and lower bound grow monotonically when tightness

is increased, but they grow with different rates. The upper bound becomes different from zero at the point where problems become overconstrained and, from that point, it grows at a more or less constant rate. The lower bound takes value zero for low tightness overconstrained problems. There is a point (around p_2=.85) where it suddenly starts growing very fast with an increasing rate. Both bounds meet at problems having the highest tightness. Therefore, we can conclude that when PFC-DAC solves easy problems on the left, it typically has low upper and lower bounds. On the other hand, when it solves easy problems on the right, it typically has high upper and lower bounds. Hard problems on the peak cause PFC-DAC to search with high upper bounds and low lower bounds.

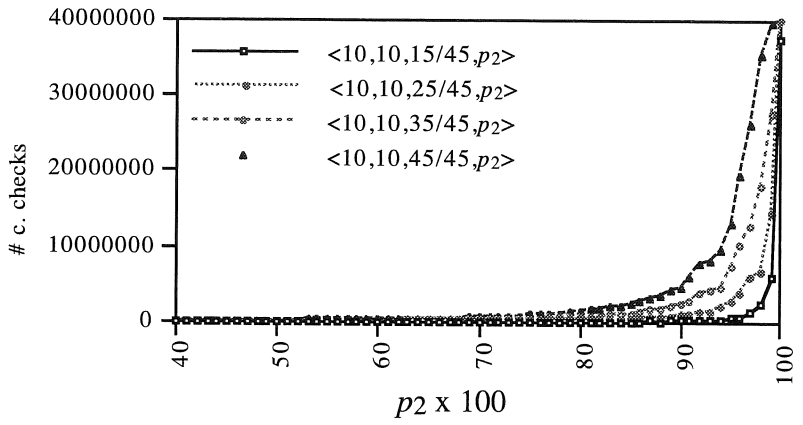


Figure 4.14: Average number of consistency checks performed by PFC with four classes of random problems.

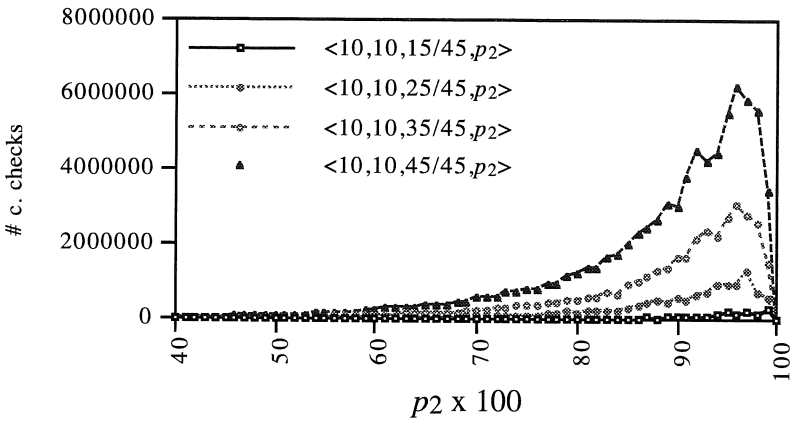


Figure 4.15: Average number of consistency checks performed by PFC-DAC with four classes of random problems.

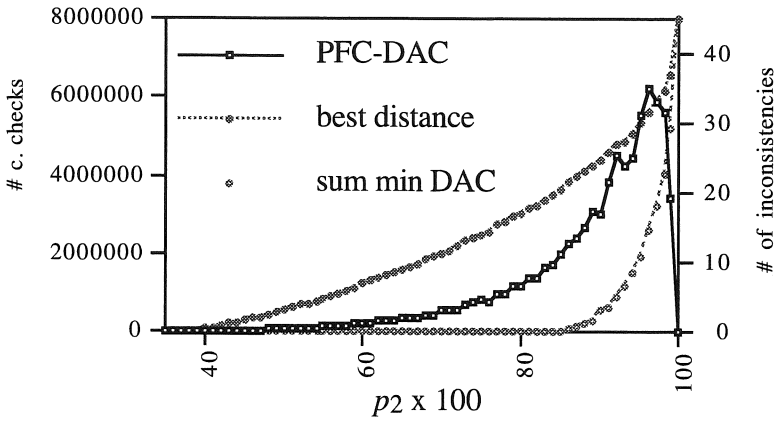


Figure 4.16: Search effort in terms of checks required by PFC-DAC plotted against evolution of selected upper and lower bounds for the <10,10,45/45,p_2> class of random problems.

4.9.2. Empirical Evaluation of the Improvements on DAC Usage.

The second set of experiments endeavours to quantify the importance of our improvements with respect to PFC-DAC. In these experiments we used the following classes of random problems,

(a). <10,10,45/45,p_2> (b). <15,5,105/105,p_2>
(c). <15,10,50/105,p_2> (d). <20,5,100/190,p_2>
(e). <25,10,37/300,p_2> (f). <40,5,55/780,p_2>

Observe that (a) and (b) are highly connected, (c) and (d) are problems with medium connectivity, and (e) and (f) are sparse problems. For each parameter setting, we generated samples of 50 instances. In this experiment, search was abandoned if the best solution was not found with 6×10^7 consistency checks.

Firstly, we analyze the individual contribution of each algorithmic enhancement presented in Sections 4.4-4.8. With this purpose we solved problem classes (a) and (e) with a sequence of algorithms that incrementally incorporate the improvements. These algorithms are:

- PFC-DAC: as described in Section 4.2.
- PFC-DAC-S4: is the algorithm that includes the lower bound introduced in Section 4.4.
- PFC-DAC-S5: is the algorithm that includes the detection of redundant constraint checks.

- PFC-GDAC: as described in Section 4.6.
- PFC-RDAC: as described in Section 4.7.
- PFC-MRDAC: as described in Section 4.8

Regarding variable ordering heuristics, algorithms requiring a static variable ordering use *forward degree* breaking ties with *backward degree* (FD/BD). The rest of algorithms use *minimum domain* breaking ties with *graph degree* (MD-DG) as dynamic variable ordering. PFC-MRDAC can use a dynamic variable ordering. However, it used (FD/BD) because we found that this static ordering was more effective than (MD-DG) for these problems. Regarding value ordering heuristics, every algorithm selected values by increasing IC+DAC. In algorithms using graph-based DAC (*i.e.*: PFC-GDAC and successors), the initial graph was decided giving to each constraint the direction that had more arc-inconsistencies.

Figure 4.17 reports the average number of consistency checks for each algorithm in the two classes of problems. In these plots we do not include PFC-MRDAC because its AC-4 based propagation to maintain DAC is performed without consistency-checks. Therefore, a comparison in terms of checks is not appropriate. It must be mentioned that, in the $<25,10,37/300,p_2>$ class, PFC-DAC reaches the upper limit of 6×10^7 consistency checks when solving problem instances in the range $0.90 \leq p_2 \leq 0.98$. Therefore, the plot is favoring this algorithm. Figure 4.17 clearly shows that each algorithm outperforms its predecessors. The algorithm improvements that have a larger impact in terms of checks are given by PFC-DAC-S4 and PFC-DAC-S5 for dense problems, and by PFC-DAC-S4 and PFC-RDAC for sparse problems. Our approach gives the most important benefit at the peak, where the hardest problems for PFC-DAC occur. The only exception to this behaviour occurs with PFC-GDAC in the tightest sparse problems. Using a dynamic variable ordering and static graph-based DAC does not produce a good performance for these problems.

Figure 4.18 reports the number of visited nodes for the same experiment. Now, we do not plot PFC-DAC-S5 because it does not produce any saving in term of nodes with respect to PFC-DAC-S4 (see Observation 4.4). Regarding visited nodes, the relative behaviour of algorithms is basically the same as with consistency checks, but gains are even larger. The best algorithm, PFC-MRDAC, practically flattens out the complexity peak. It can be observed that maintaining DAC (PFC-MRDAC), which was not considered in the previous plot, produces a significant tree reduction with respect PFC-RDAC.

Finally, Figure 4.19 reports average CPU time (it includes every algorithm). It can be observed that PFC-DAC-S5 only has a positive effect in the tightest dense problem. This fact may be surprising because PFC-DAC-S5 gains in terms of checks were considerable. The reason is that performing a consistency check in a random problem is only a table look-up and it can be done very efficiently. Thus, this improvement is almost unnoticeable for random problems. We believe that its significance will

become more apparent in domains where consistency checks are more costly. More importantly, it can be observed that maintaining DAC (PFC-MRDAC) is not cost effective for dense problems. Although it produces important tree reductions (Figure 4.18), the overhead required does not pay off.

Additional experiments aimed at a more exhaustive evaluation of PFC-GDAC, PFC-RDAC and PFC-MRDAC with respect to PFC-DAC. With this purpose, the six classes of problems were solved with these four algorithms. As in the previous experiment, the static and dynamic variable orderings were FD/BD and MD/DG, respectively. Values were dynamically ordered by IC+DAC. Like before, PFC-MRDAC uses the static variable ordering because it was found more effective. In this experiment CPU time is taken as the main search effort measurement because both PFC-RDAC and PFC-MRDAC perform an important overhead which does not perform any consistency check. The number of visited nodes is used to evaluate the impact of the algorithms to anticipate dead-ends, regardless of the computational cost of computing their bounds.

Figures 4.20, 4.21 and 4.22 report CPU time for dense, medium and sparse problems, respectively. It must be mentioned that PFC-DAC reaches the established search effort limit and abandons search before completing its traversal in several executions with the medium connectivity and sparse problems. Therefore, the gains given by our improvements are even larger than the ones shown in the plots.

Regarding PFC-GDAC, we can observe that it clearly outperforms PFC-DAC, except for the tightest sparse problems, where the use of a dynamic variable ordering heuristic does not seem to be suitable. The largest gain occurs in dense problems where PFC-GDAC is more than 12 times faster than PFC-DAC. We still have not found a satisfactory explanation for the bad performance that PFC-GDAC has on tight sparse problems. PFC-RDAC improves PFC-DAC and PFC-GDAC in all problem classes. The gain grows with problem tightness. PFC-RDAC is more than 4,000 times faster than PFC-DAC on the tightest sparse problems. Regarding PFC-MRDAC, we observe that maintaining RDAC only pays off on the tightest instances and on the most sparse problems. Since Figure 4.22 does not show the relative performance of PFC-MRDAC with respect PFC-RDAC clearly, we present in Figure 4.23 the average CPU time of these two algorithms on sparse problems. It can be observed that maintaining DAC on sparse problems produces a significant gain.

Figure 4.24-4.26 report the number of visited nodes for dense, medium and sparse problems. We observe a similar behaviour as in terms of time. The only exception is the performance of PFC-MRDAC which produces an important tree reduction with respect to PFC-RDAC —even with dense and medium connectivity problems. Therefore, maintaining DAC anticipates dead-end detection, but it is too costly to pay-off in some classes of problems. For this reason, we believe that better results will be

obtained with a more sophisticated AC algorithm (*i.e.*: AC-7 [Bessière *et al.*, 95]).

4.10 Conclusions and Future Work

From the work presented in this Chapter, we can conclude that the use of local consistency information is a major step forward in the development of efficient branch and bound algorithms for MAX-CSP. We have shown that even the simplest use of DAC may produce an exponentially large gain in PFC. We have examined a complexity peak in MAX-CSP when solving random problems. This phenomenon, not reported before, seems to be related to the quality of branch and bound lower bound. Furthermore, we have developed three new algorithms that improve PFC-DAC in a number of aspects. These new algorithms are among the best existing approaches for MAX-CSP.

This work also leaves some questions without an answer. Regarding the complexity peak, a deeper analysis focusing on a good characterization of the peak is of obvious interest. Considering the new algorithms, we have found quite surprising that introducing dynamic variable orderings does not always give an improvement over static variable orderings. This contradicts traditional wisdom in total constraint satisfaction and requires further analysis. It is our belief that lower bound quality is the major issue in branch and bound algorithms for partial constraint satisfaction. Considering the simplicity of the greedy optimization algorithm presented in Section 4.7, we think that there is still room for further improvements. Regarding the algorithm that maintains RDAC updated, we believe that it can also be improved by moving to more advanced AC schemas. Finally, an exhaustive comparison between DAC-based algorithms and Russian Doll Search still remains to be done.

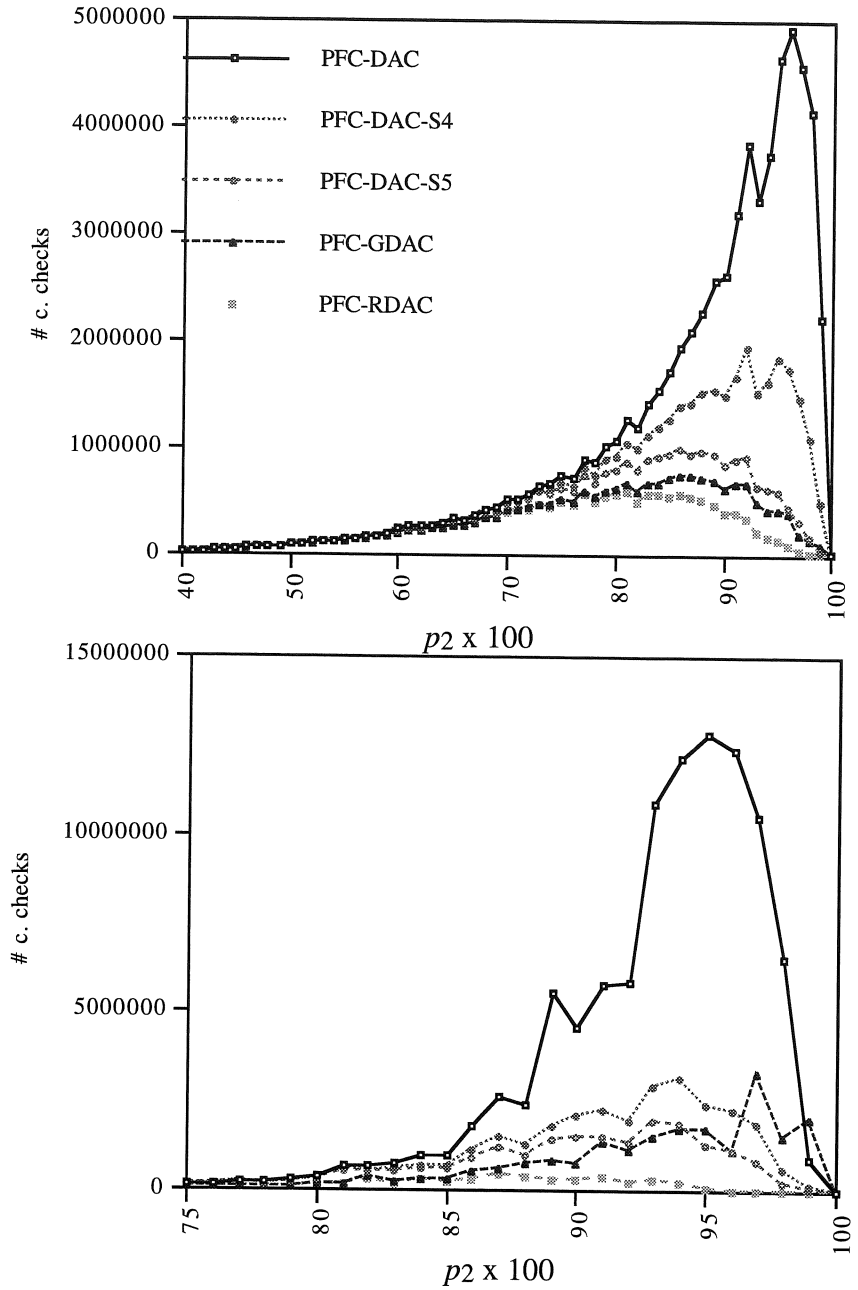


Figure 4.17: Average number of consistency checks of different algorithms on the <10,10,45/45,p_2> and <25,10,37/300,p_2> classes of problems.

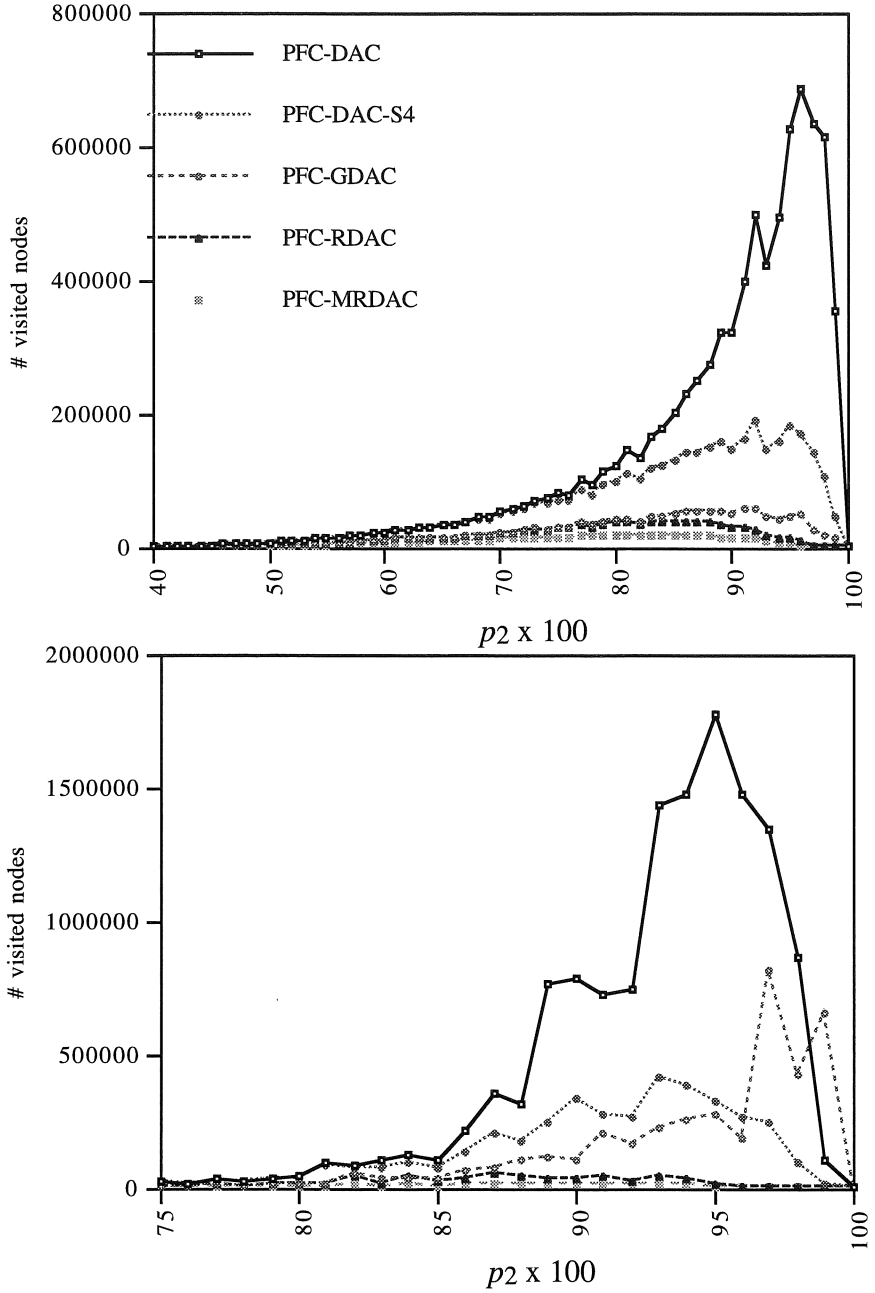


Figure 4.18: Average number of visited nodes of different algorithms on the <10,10,45/45,p_2> and <25,10,37/300,p_2> classes of problems.

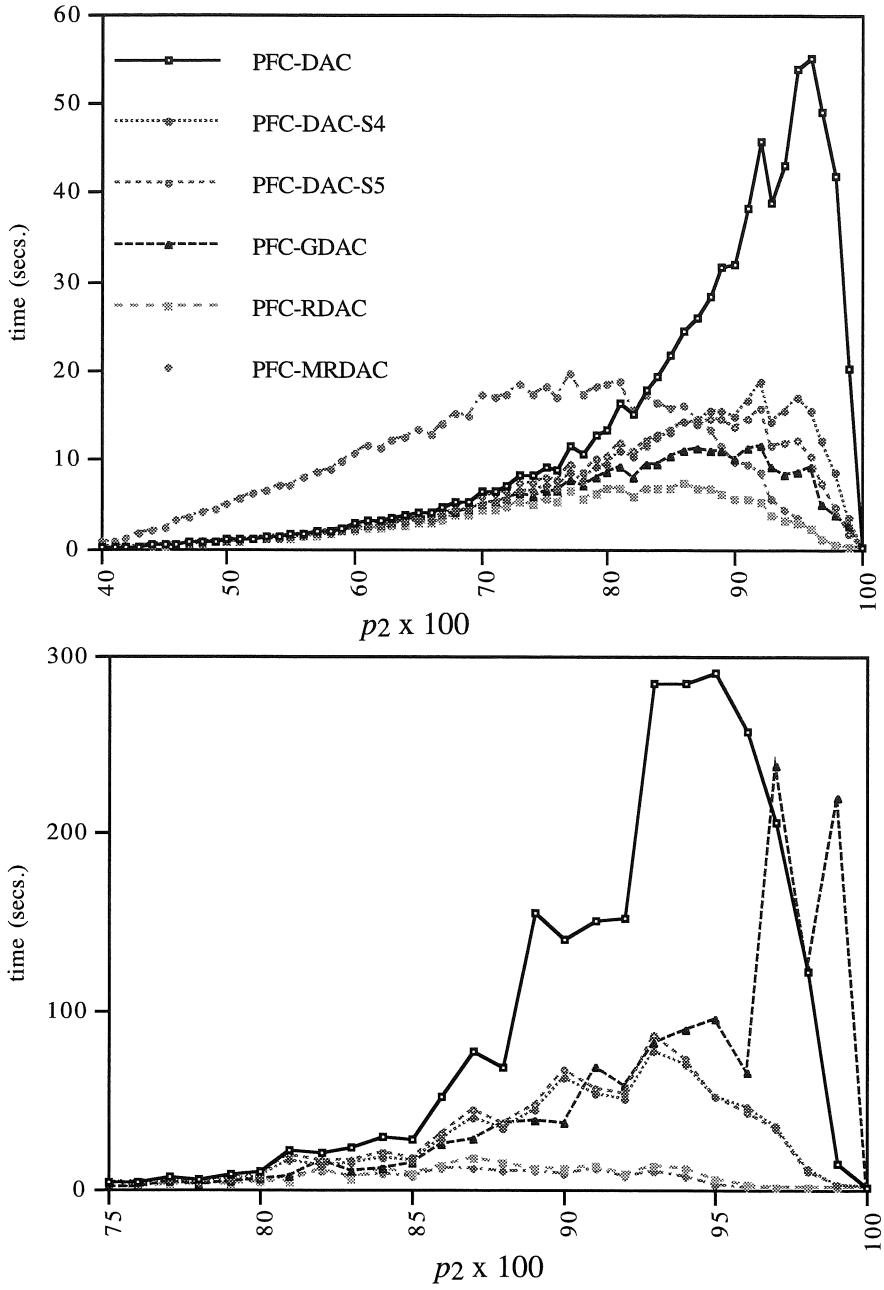


Figure 4.19: Average CPU time for different algorithms on the <10,10,45/45,p_2> and <25,10,37/300,p_2> classes of problems.

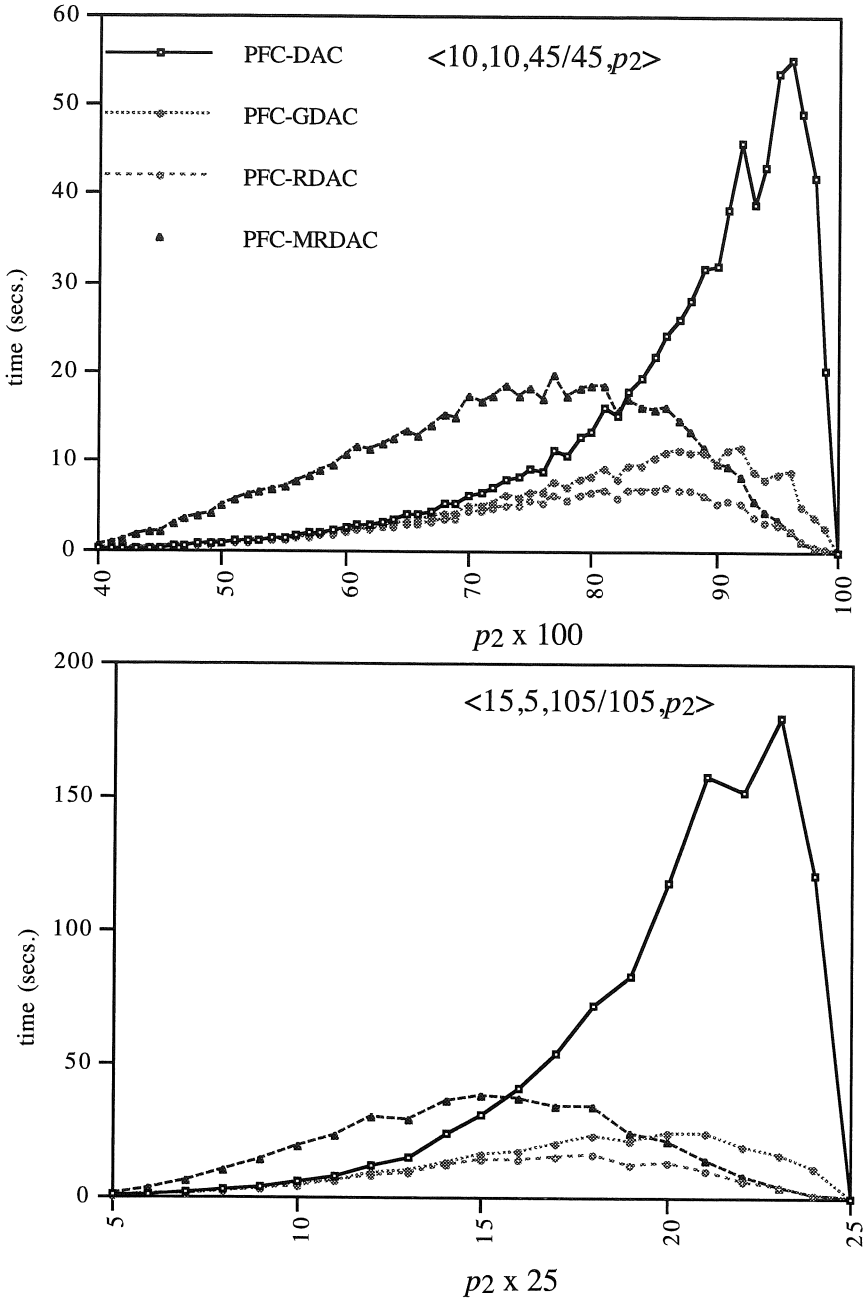


Figure 4.20: Search effort in terms of CPU time for different algorithms and two classes of dense problems.

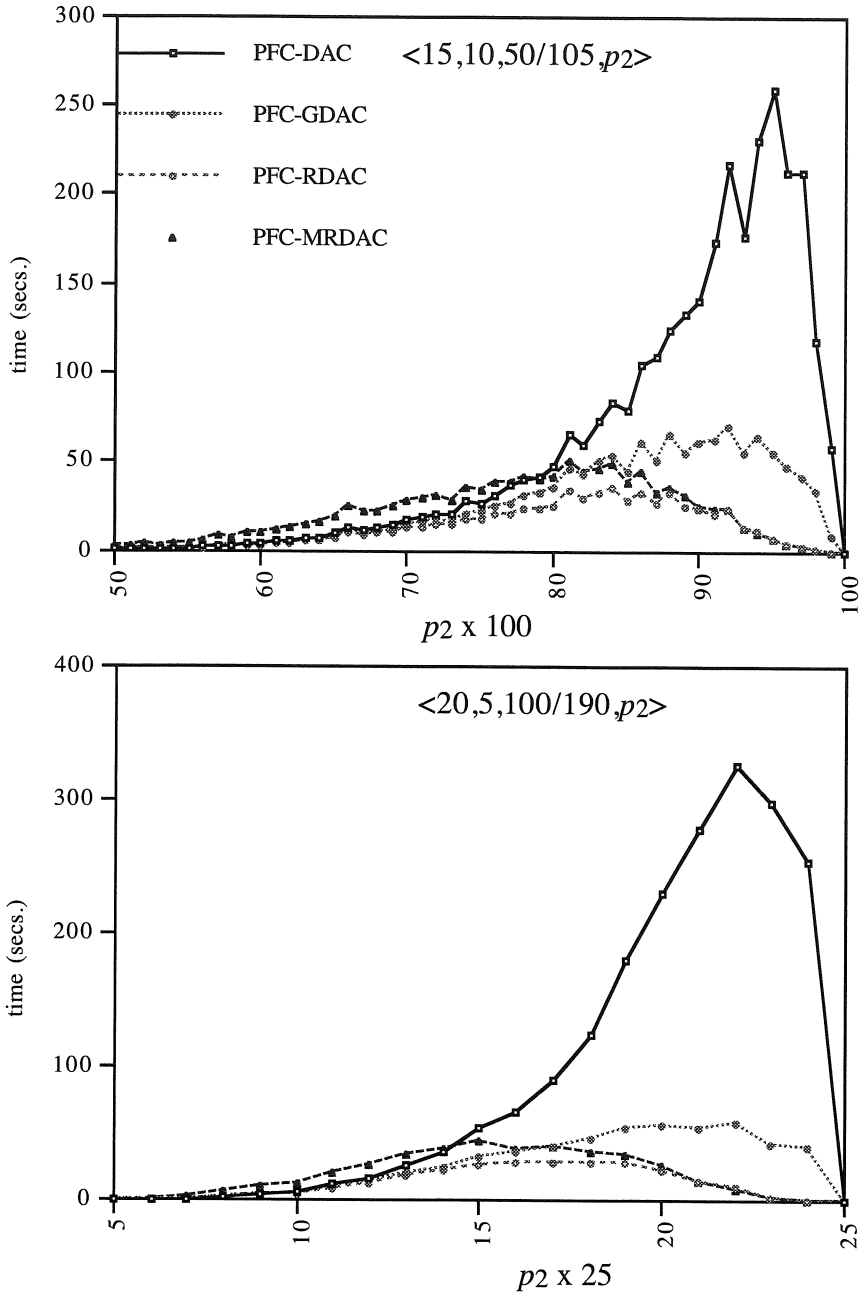


Figure 4.21: Search effort in terms of CPU time for different algorithms and two classes of medium connectivity problems.

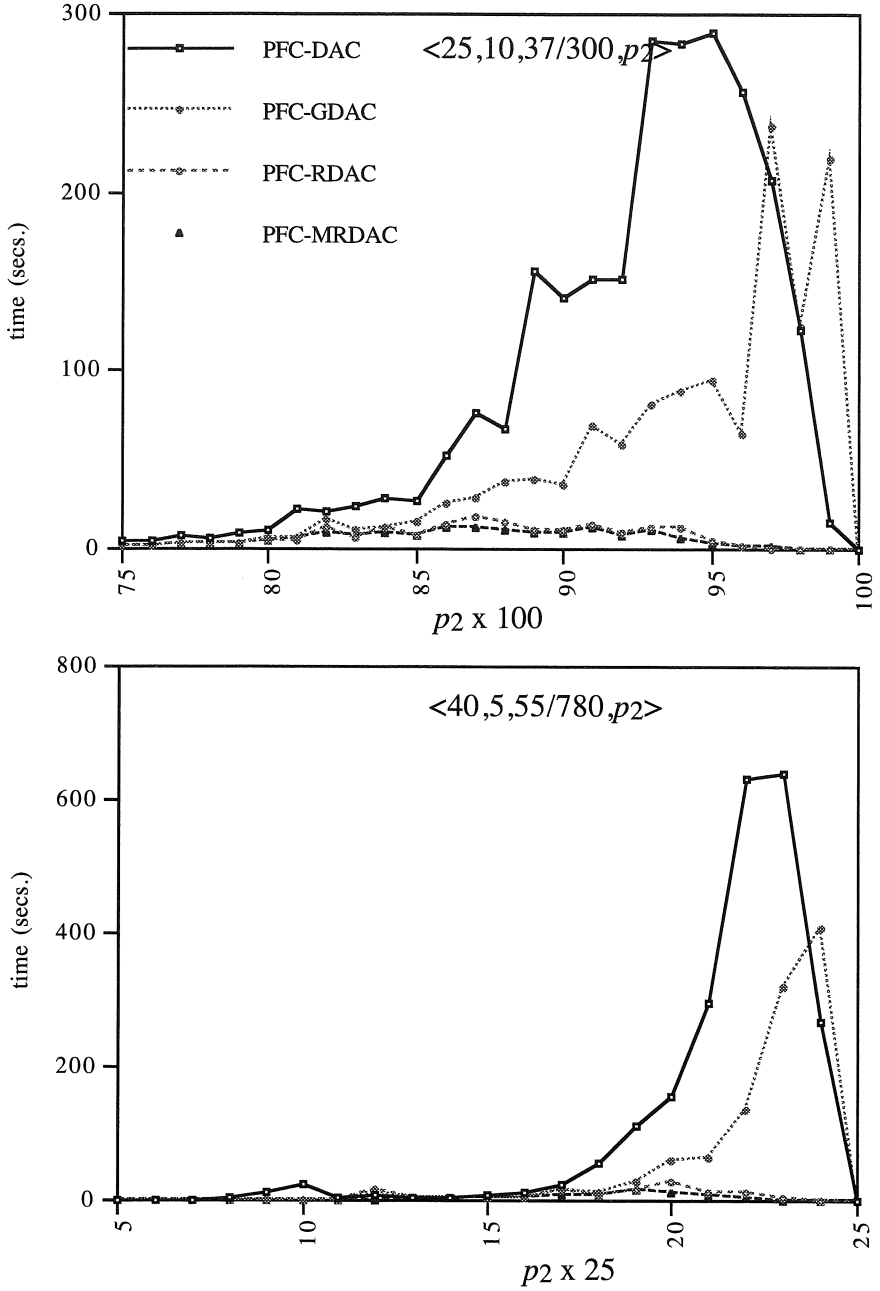


Figure 4.22: Search effort in terms of CPU time for different algorithms and two classes of sparse problems.

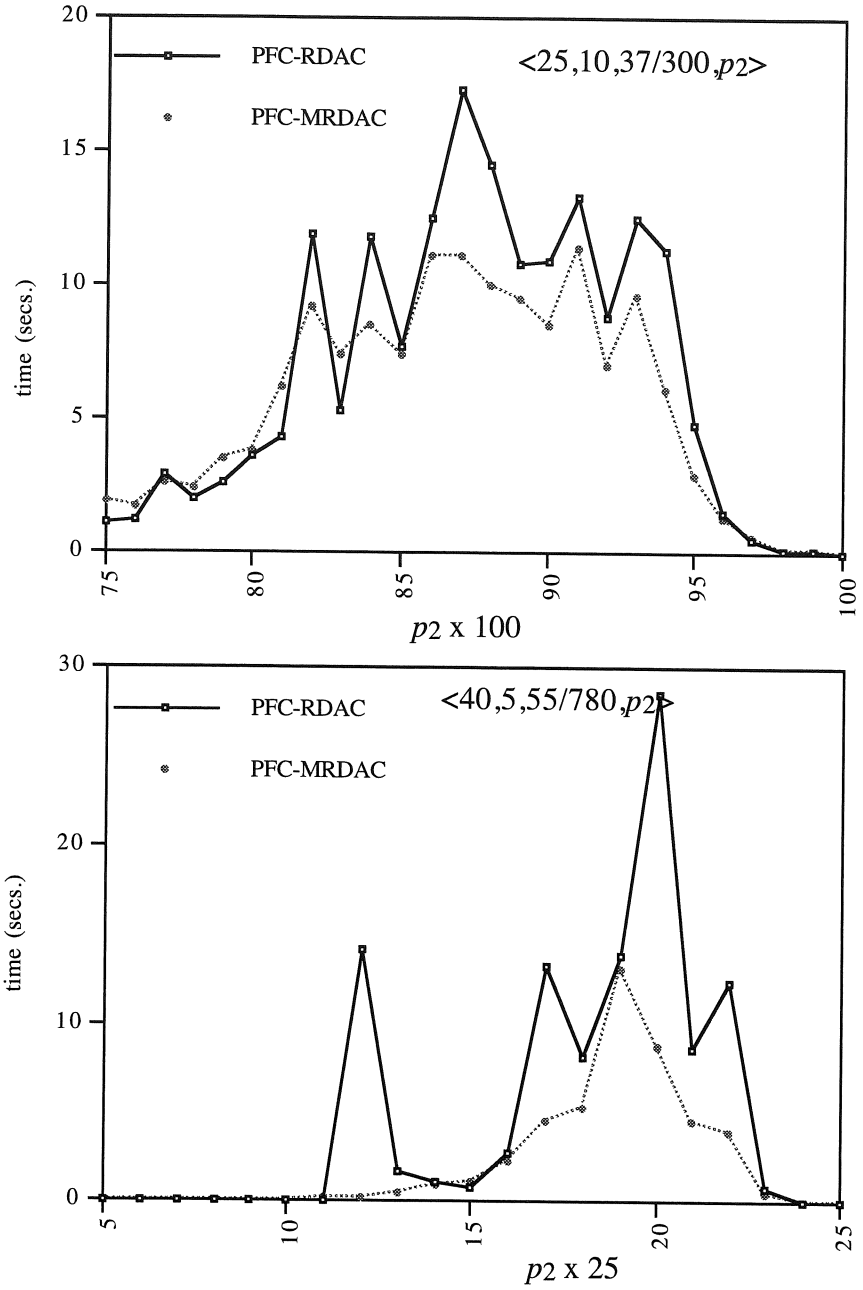


Figure 4.23: Experimental results of PFC-RDAC and PFC-MRDAC on two classes of sparse problems.

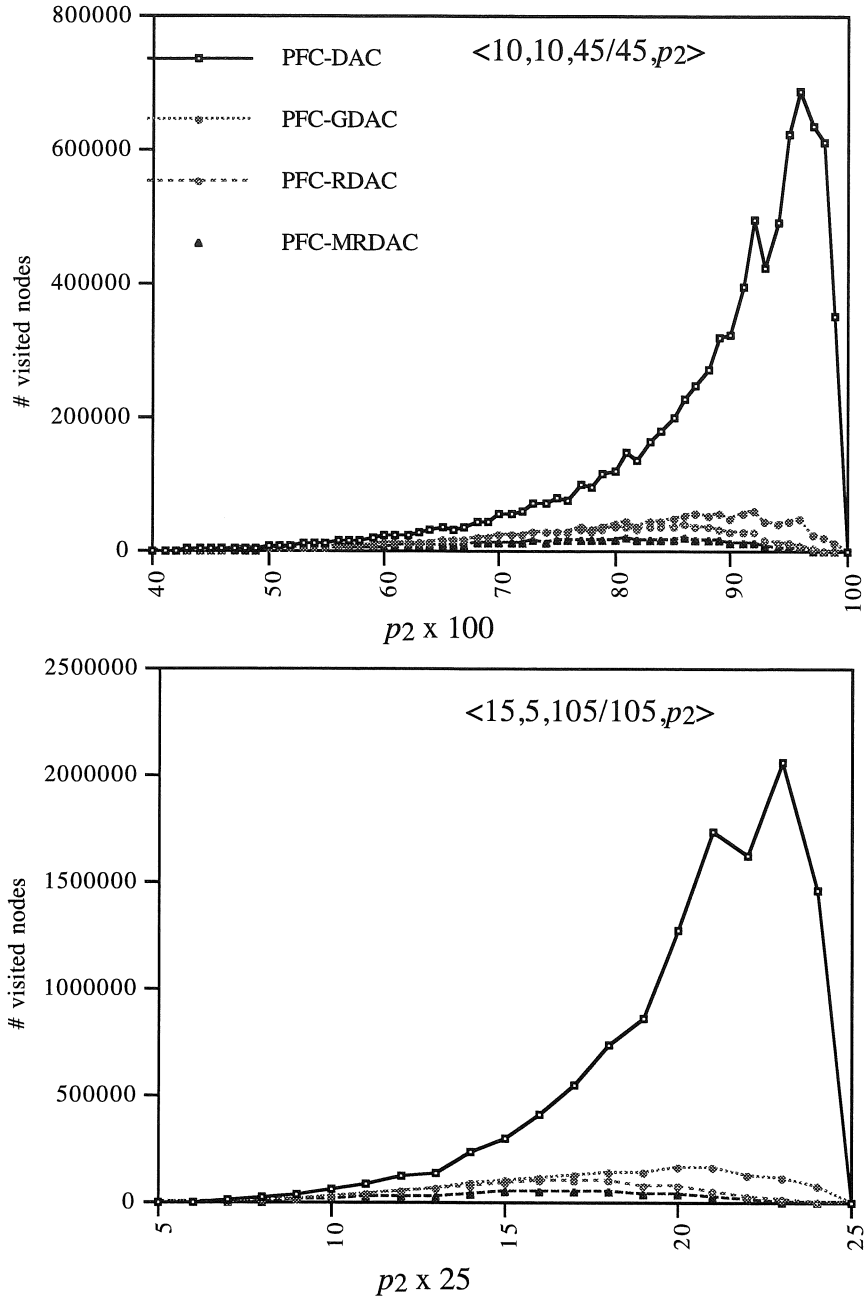


Figure 4.24: Average number of visited nodes for different algorithms and two classes of dense problems.

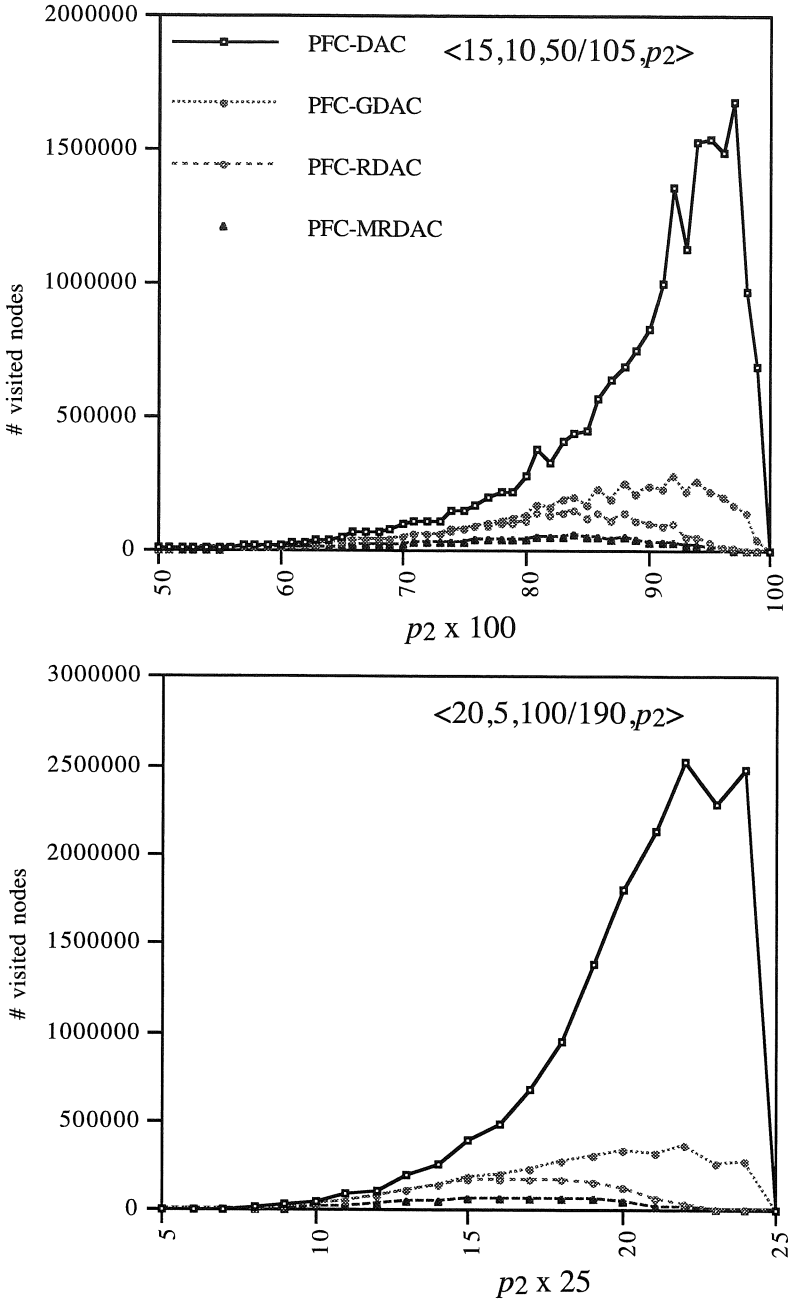


Figure 4.25: Average number of visited nodes for different algorithms and two classes of medium connectivity problems.

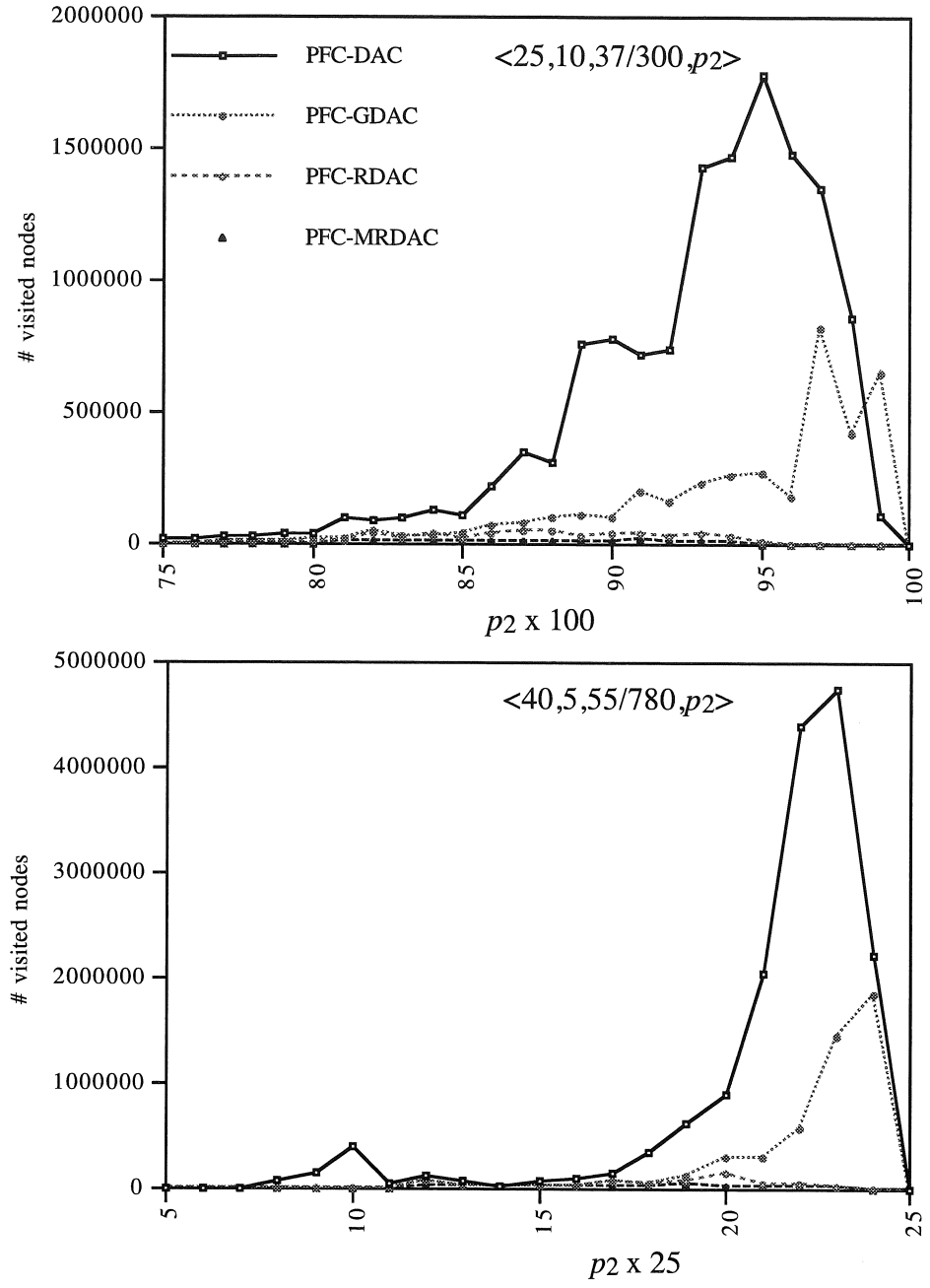


Figure 4.26: Average number of visited nodes for different algorithms and two classes of sparse problems.

Chapter 5

Lazy Evaluation in Partial Constraint Satisfaction

In the previous Chapter, we showed that combining search with local consistency enforcement is an effective technique for early dead-end detection. In general, higher levels of local consistency have better dead-end detection capabilities at the cost of performing more computation at each node. Obviously, the overhead of propagation has to be outweighed by its gain. Since the key issue is the trade-off between the propagation cost and the gains that it brings in the form of tree reductions, it is of much practical importance to develop efficient methods for propagation.

Lazy evaluation is a general algorithmic technique which consists in delaying computation until it is strictly necessary. As a result, no redundant computations are done at the extra cost of more complex algorithms. In the total constraint satisfaction context, lazy evaluation has been successfully applied to avoid performing redundant consistency checks. In this Chapter, we explore the same idea in partial constraint satisfaction. We show that this approach is even more appropriate in this context because algorithms for partial constraint satisfaction have more sources of redundant computation. We present a lazy algorithm which saves a good deal of consistency checks when computing the dead-end condition at each visited node. We show that this approach can be naturally combined with other algorithms presented in this work. Thus, their efficiency can be joined. Our experiments give support to the suitability of the approach.

This Chapter is organized as follows. In Section 5.1, we give an introduction. In Section 5.2, we revise previous work on lazy evaluation in the total constraint satisfaction context. In Section 5.3, we introduce a lazy algorithm for MAX-CSP and justify that it never performs more consistency checks than its non lazy counterpart. In Section 5.4, we provide experimental evidence of the gains that it produces, combined with both

DAC and RDS. Finally, in Section 5.5, we give the conclusions of the Chapter.

5.1 Introduction

As we have seen in the previous Chapter, combining search with local consistency enforcement is an effective technique for early dead-end detection. Different forms of arc-consistency can be successfully applied in solving both total and partial constraint satisfaction. In general, higher levels of local consistency have better dead-end detection capabilities at the cost of performing more computation at each node. Obviously, the overhead of propagation has to be outweighed by its gain, which becomes apparent in the form of tree reductions. On very easy problems, even the simplest forms of propagation can be useless because solutions are easily found, or dead-ends are easily detected. But, on hard problems it is clearly cost-effective to invest some additional effort at each node which produces an overall gain.

A good example is found in the total constraint satisfaction context. Historically, simple backtracking was used for CSP solving. The additional computational effort of forward checking look-ahead was rapidly accepted to be cost effective for most non-trivial problems. Lately, it has been shown that MAC, which performs more propagation than FC, can outperform FC on sufficiently hard problems. The acceptance of MAC superiority for hard problems has needed: (*i*) the development of very efficient arc-consistency algorithms (*i.e:* AC-7 [Bessière et al., 95]), and (*ii*) the use of hard random problems to benchmark algorithms.

In the partial constraint satisfaction context, there is a clear parallelism. PFC outperforms look-back schemas for most problem instances. More than that, we showed in Chapter 4 that the extra cost of adding arc-consistency information to PFC is, with no doubt, advantageous. The additional effort of maintaining directed arc-inconsistency counts updated (MDAC) is beneficial for some very hard overconstrained instances.

Since the key issue is the trade-off between the propagation cost and the gains that it brings in the form of tree reductions, it is of much practical importance to develop efficient methods for propagation.

Lazy evaluation is a general algorithmic technique. The idea behind it is to delay computation until it is strictly necessary. As a result, no redundant computations are done at the extra cost of more complex algorithms. Lazy evaluation has been applied to constraint satisfaction in *Minimal Forward Checking* (MFC) [Zweben and Eskey, 89; Dent and Mercer, 94; Bacchus and Grove, 95] and in *Lazy Arc Consistency* [Schiex *et al.*, 96]. In both cases, the corresponding non-lazy algorithms were doing more than

needed to achieve their goals, so introducing this technique caused efficiency improvements. Regarding MFC, it was realized that FC look-ahead was checking every value of every future domain, which is more than needed to detect empty domains. When a future domain has at least one consistent value it is not empty, so look-ahead can stop and continue on another future domain. MFC may save consistency checks when performing look-ahead on domains of future variables whose assignment is never attempted.

In this Chapter, we apply lazy evaluation to partial forward checking obtaining a new MAX-CSP algorithm called *partial lazy forward checking* (PLFC). This new algorithm is substantiated by the fact that PFC look-ahead checks every value in every future domain, which is more than really needed to detect dead-ends. As long as one value is not pruned its domain is not empty and it does not cause a dead-end, so look-ahead can stop there and continue on another future domain. The best candidate to remain is the value with minimum IC in its domain, so computing this minimum IC is the amount of look-ahead required on that domain for dead-end detection. If the value with minimum IC is pruned, all other values must be pruned as well, so the domain becomes empty and a dead-end is detected. If the value with minimum IC is not pruned, the domain is not empty and look-ahead can continue on another domain. PLFC may save consistency checks when performing look-ahead on domains of future variables whose assignments are never attempted. PLFC does not prune future values, except the one with minimum IC in its domain (whose pruning means a dead-end). A value is tested for pruning when it is assigned to the current variable. Interestingly, delaying value pruning until current variable assignment causes it to use bounds better than or equal to the ones used by PFC for future value pruning, which may save consistency checks. While savings in look-ahead were expected (because of the parallelism with FC), savings because of pruning delay is a new effect not reported previously which reinforces the performance improvement of the lazy approach. PLFC advantage is proven theoretically, showing that in terms of checks PLFC never performs worse than PFC with the same variable and value ordering. Interestingly, PLFC can be directly combined either with DAC or RDS approaches producing very efficient algorithms. Experimental results show clear performance improvements in number of checks and in CPU time.

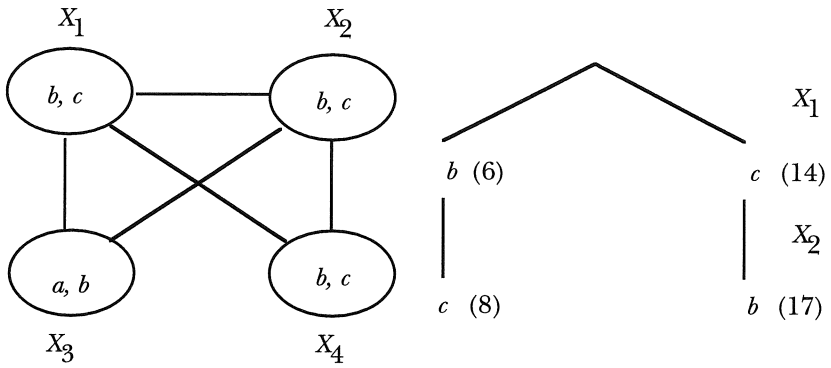
5.2 Previous Work

5.2.1 Forward Checking Redundancies

As we have mentioned, FC look-ahead strategy causes some of the consistency checks that it performs never to be used. This situation occurs when FC tries to prune a value of a variable whose assignment is never attempted because the algorithm always backtracks before the variable is chosen as the current variable.

Example 5.1:

The following picture represents a graph coloring problem and the search tree that FC traverses to solve it. At each tree node, we include the number of consistency checks that FC has performed up to this point, including the current propagation.



This particular problem is unsolvable because there is no consistent assignment including variables $\{X_1, X_2, X_4\}$. Assigning the first two variables always produces an empty domain in D_4. Observe that all consistency checks associated with $b \in D_3$ are redundant in the sense that the algorithm never uses them. The reason is that all propagations find value a $\in D_3$ consistent, so when the feasibility of $b \in D_3$ is checked, it is already known that D_3 will not become empty. In addition, the assignment $X_3 \leftarrow b$ is never attempted because dead-ends are always detected at the second tree level so its consistency with respect to past variables is never needed.

FC needs to perform 17 consistency checks to solve the problem. Three of them are redundant checks associated with $b \in D_3$ ($(b,b) \in R_{13}$, $(c,b) \in R_{13}$ and $(c,b) \in R_{23}$)

This situation in which FC performs redundant consistency checks is formalized in the following observation.

Observation 5.1:

Let S be a search state where $X_i \leftarrow a$ is the current assignment, and b is a feasible value of a future variable X_j. Assume that FC performs look-ahead and no domain becomes empty. Consequently, all consistency checks between the current assignment and feasible future values have been performed. In particular, the consistency of $(a,b) \in R_{ij}$ is checked. Suppose that FC proceeds its search below in the tree and finally backtracks to S satisfying the following two conditions:

- X_j has never become the current variable.
- for all detected dead-ends, the domain of X_j had at least one feasible value before b.

In this situation, the consistency check $(a,b) \in R_{ij}$ performed in S and all subsequent checks associated with value b in successor nodes of S could have been avoided because the algorithm did not make any use of them.

5.2.2 Lazy Forward Checking

The inefficiency associated to FC greedy pruning was first reported in [Zweben and Eskey, 89]. In their paper, they present a lazy version of FC that avoids doing consistency checks until they are absolutely necessary. The same algorithm was independently developed by Dent and Mercer. They called it *minimal forward checking* (MFC) and described it in detail in [Dent and Mercer, 94]. Finally, in [Bacchus and Grove, 95] MFC is again considered and additional features about its modus operandi are given.

MFC improves FC exploiting the fact that not all the pruning effort that FC performs is strictly necessary to determine the existence of a dead-end. After each assignment, FC checks all future values pruning those values which are inconsistent with it. If during this process an empty domain occurs, the algorithm backtracks. Applying the notion of lazy evaluation, all we need to compute during the propagation is whether an empty domain occurs or not. Thus, we do not need to check every value. It is enough to detect one consistent value for each future variable. The rest of the checks can be delayed until they are strictly necessary. If it turns out that a dead-end is detected before performing them, we will not have wasted effort on those checks.

However, with this lazy technique, when a new assignment is considered we are no longer sure that the current value is consistent with past assignments. Therefore, one needs to guarantee the consistency of the current assignment performing backward consistency checks. A naive

implementation would compute this by checking consistency with all past variables. But some of this backward consistency may have been previously proven as part of the lazy look-ahead when the current variable was still a future variable. In order to avoid the computation of redundant consistency checks, MFC follows a *bookkeeping* strategy similar to that of *backmarking* [Gaschnig, 77]. It keeps a table, *cons_level*, that records information on previously computed consistency. More precisely, if b is a feasible value of a future variable X_j, then $cons\_level_{jb} = v$ means that consistency of value b has been already proven with the first v past variables and their current assignment. If its consistency has to be checked, it only has to be checked against past variables after its *cons_level* count.

MFC appears in Figure 5.1. For simplicity reasons, it is assumed that variables are assigned in lexicographical order. Consequently, the set of past and future variables can be replaced by an index i indicating the current variable. In this algorithm, we also assume that domains and the *cons_level* data structure are global variables. At each search state, MFC selects the current variable and considers the assignment of all its feasible values. When value a is assigned to X_i, MFC performs a two-step procedure. First, it checks its consistency with respect to past variables after $cons\_level_{ia}$ (line 8). Second, if the previous step is successful, it computes the minimal amount of checks to establish the empty domain condition (line 9). Thus, MFC can be seen as a hybrid strategy combining look-back with look-ahead. The look-back part is similar in spirit to *backmarking* [Gaschnig, 77]. The look-ahead part is similar to forward checking but following the lazy approach.

In our implementation, *lazy_look_back*(k, c, max) checks the consistency of $X_k \leftarrow c$ with respect to past variables between its level of established consistency, $cons\_level_{kc}$, and max. If the test succeeds, then $cons\_level_{kc}$ takes value max and the function returns *true*. If the test fails because $X_k \leftarrow c$ is inconsistent to a past variable X_w, then $cons\_level_{kc}$ takes value w, value c is pruned from D_k (it will not be restored until the current assignment to X_w is changed), and finally the function returns *false*.

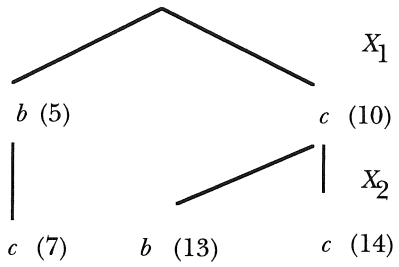
The procedure *lazy_look_ahead*(i, a) performs the lazy propagation of $X_i \leftarrow a$. It iterates on the set of future variables, searching for one value in their domain being consistent with past variables. For each value that it considers, it uses *lazy_look_back*() to *catch up* its possibly delayed consistency checks to past variables. This function returns *true* if no empty domain is detected, *false* otherwise.

Finally, *restore*(i, a) is in charge of domain restoration upon backtracking. It retrieves all values of future domains that were pruned because of their inconsistency with the current assignment $X_i \leftarrow a$. It also updates *cons_level* because the current assignment is going to be changed and all future values whose consistency was checked up to variable X_i are no longer updated. For the sake of simplicity in the code, our implementation performs consistency checks to retrieve information. However, the same effect but in a consistency check free manner can be obtained by keeping a

list of pruned values associated with each past variable. In the following we assume the consistency check free implementation.

Example 5.2:

Consider the graph coloring problem of the previous example. If MFC solves it, the following search tree is obtained.



We indicate the accumulated number of checks performed at each node, including the propagation of the current assignment. After the first assignment $\{X_1 \leftarrow b\}$, MFC look-ahead leaves the following domains: $D_2 = \{c\}$, $D_3 = \{a, \underline{b}\}$ and $D_4 = \{c\}$ (we underline values that may not be feasible with respect to all past variables because their consistency is not updated). After the second assignment $\{X_1 \leftarrow b$, $X_2 \leftarrow c\}$ MFC look-ahead leaves the following domains: $D_3 = \{a, \underline{b}\}$ and $D_4 = \emptyset$. So, a dead-end is detected without requiring the delayed test associated with $b \in D_2$ ($(b,b) \in R_{13}$), so it has been avoided. Next, MFC backtracks to the first level and attempts a new assignment $\{X_1 \leftarrow c\}$. Look-ahead leaves the following domains: $D_2 = \{b, \underline{c}\}$, $D_3 = \{a, \underline{b}\}$ and $D_4 = \{b, \underline{c}\}$, where several tests are delayed. Since there is not empty domain, a new assignment $\{X_1 \leftarrow c, X_2 \leftarrow b\}$ is attempted. Its look-ahead retrieves some delayed tests (all except $(c,b) \in R_{13}$) and leaves the following domains: $D_3 = \{a, \underline{b}\}$ and $D_4 = \emptyset$. Because of the empty domain, MFC backtracks and attempts $\{X_1 \leftarrow c, X_2 \leftarrow c\}$ but when catches up its delayed check with X_1, it is detected unfeasible. Since there are no more choices, search is abandoned. Note that FC performs redundant checks associated with $b \in D_2$ (see Example 5.1), which are skipped by MFC.

```
function MFC (i, Assg) returns boolean
1       if (i=n+1) then
2               Sol:= Assg
3               return(true)
4       endif
5       stop:= false
6       for all a ∈D_i while (not stop) do
7               NAssg:= Assg ∪ {X_i←a}
8               if (lazy_look_back(i, a, Assg, i-1)) then
9                       if (lazy_look_ahead(i, a, NAssg)) then
10                              stop := MFC(i+1, NAssg)
11                      endif
12                      restore(i, a)
13              endif
14      endfor
15      return(stop)
endfunction
function lazy_look_back (k, c, Assg, max) returns boolean
16      for j:= cons_level_kc +1 to max do
17              cons_level_kc:= j
18              (X_j ←v^j) := consult(Assg,j)
19              if (inconsistent(X_k←c, X_j ←v^j) then
20                      D_k:= D_k - {c}
21                      return (false)
22              endif
23      endfor
24      return (true)
endfunction
function lazy_look_ahead(i, a, NAssg) returns boolean
25      for j:=i+1 to n do
26              exit:=false
27              for all b ∈D_j while (exit=false) do
28                      if (lazy_look_back(j, b, Assg, i)) then exit:=true endif
29              endfor
30              if(D_j = ∅) then return (false) endif
31      endfor
32      return (true)
endfunction
procedure restore(i, a)
33      for j:=i+1 to n do for b:=1 to m  do
34              if (cons_level_jb=i) then
35                      if (inconsistent(X_j ←b, X_i ←a) then
36                              D_j := D_j ∪{b}
37                      endif
38                      cons_level_jb:=i-1
39              endif
40      endfor
endprocedure
```

Figure 5.1: Minimal forward checking (MFC) [Zweben and Eskey, 89; Dent and Mercer, 94; Bacchus and Grove, 95].

5.2.3 Theoretical Results

In the following, we analyze the behaviour of MFC comparing it with FC. The main result, due to [Dent and Mercer, 94], shows that MFC is never worse than FC in terms of consistency checks. To make it more comprehensive in our context, and extensible to the partial constraint satisfaction case, we make our own proof. Consider first the following observation.

Observation 5.2:

A node visited by FC or MFC is defined by a set of past variables and its current assignment. From the algorithmic description one knows that:

1. FC visits a node if and only if:
   - past variables are consistent among them,
   - the current assignment is consistent with past variables,
   - the propagation of the past variables does not produce any empty domain.
2. If MFC visits a node, then:
   - past variables are consistent among them,
   - the propagation of the past variables does not produce any empty domain.

From 1 and 2, it is easy to see that if FC and MFC solve the same problem with the same variable and value ordering, then:

3. If MFC visits a node, then its parent is visited by FC, too.

Theorem 5.1:

For any CSP, assuming the same variable selection and value selection orders, the number of consistency checks performed by MFC is lower than or equal to the number of consistency checks performed by FC.

Proof:

Consider an arbitrary CSP solved by both FC and MFC. Let $Checks(\text{FC})=\{ch_1, ch_2,...,ch_r\}$ and $Checks(\text{MFC})=\{ch'_1, ch'_2,...,ch'_{r'}\}$ the set of checks performed by FC and MFC when solving the problem, respectively. Each check is defined by a tuple (X_p,v,X_q,w,S) which means that the test was $(v,w)\in R_{pq}$ and it occurred at node S. The node is required to distinguish the different times that FC or MFC test the consistency of the same pair of values (since we are concerned with the *number of checks*, we consider them as different).

The proof has the following steps. First, we show that there is an application from *Checks*(MFC), to *Checks*(FC). Second, we show that this application is injective (namely, two different MFC checks cannot be associated to the same FC check). Third, we show that this application is not exhaustive (namely, there are some FC checks to which no MFC check is associated). Thereafter, the proof is complete. For the sake of clarity and without loss of generality, we assume that variables and values are always selected in lexicographical order.

1. *Definition of the application:* Consider an arbitrary node, S, being visited by MFC where the assignment $X_i \leftarrow a$ is being attempted and an arbitrary consistency check, $(v,w) \in R_{pq}$, is performed ($p<q$). There are two possible places where this constraint check can be performed:
   - During the lazy look-back: in this case $X_p \leftarrow v$ is an assignment made to a past variable, q is equal to i and w is equal to a. In this case this check is defined by $(X_p,v,X_i,\ a,S)$ and corresponds to a unique consistency check $(X_p,v,X_i,\ a,S')$ performed by FC. S' is the ancestor of S where X_p is the current variable. We know that FC visits S' because all nodes with successors visited be MFC are also visited by FC (Observation 5.2.3). We know that FC performs that check because it does not detect a dead-end (otherwise, MFC would not visit S) and FC performs the whole propagation when it does not detect dead-ends.
   - During the lazy look-ahead: in this case there are two additional possibilities.
     - It is a forward check: then p is equal to i, v is equal to a, q is a future value and w is one of its feasible values. In this case this checks is (X_i,a,X_q,w,S) and corresponds to a unique check (X_i,a,X_q,w,S) performed by FC during its look-ahead at node S. We know that FC visits S because if MFC performs look-ahead, the current assignment has been successfully checked with all past variables, then all the conditions of FC for visiting S are fulfilled (Observation 5.2.1). We know that FC performs that check because, due to its lazy strategy, the forward checks that MFC performs are a subset of the forward checks that FC performs at the same node.
     - It is a backward check (performed during a call to the lazy look-back function associated to a future value): then $X_p \leftarrow v$ is an assignment made to a past variable, q is a future value and w is one of its values. In this case this check is (X_p,v,X_q,w,S) and corresponds to a unique check (X_p,v,X_q,w,S') performed by FC. S' is the

ancestor of S where X_p is the current variable. Again, we know that FC visits S' because all nodes with successors visited be MFC are also visited by FC. We know that FC performs that check because it does not detect a dead-end and FC performs the whole propagation when it does not detect dead-ends.

2. *Proof of injectivity*: Suppose that the application associates two different MFC checks, (X_p,v,X_q,w,S') and (X_p,v,X_q,w,S''), with the same FC check (X_p,v,X_q,w,S). Without loss of generality we take (X_p,v,X_q,w,S') to be check that occurs before chronologically. Because of the way the application is built, we know that S' and S'' belong to the subtree rooted by S. When MFC performs the check (X_p,v,X_q,w,S), it sets $cons\_level_{qw}$ to value p. MFC cannot perform any other check associated with the same pair of values until it backtracks up to X_p (see procedure *restore* in Figure 5.1). Therefore, MFC does not perform a check (X_p,v,X_q,w,S'') such that S'' is in the subtree rooted by S.
3. *Proof of non exhaustivity:* Consider a problem whose first solution in lexicographical order is the assignment of the first value to all its variables. In this case, FC performs more checks than MFC. Therefore, the application is not exhaustive.

5.2.4 Practical Significance

From the previous analysis, we know that MFC never performs worse than FC regarding consistency checks. Still, it does not indicate the magnitude of the improvement. Experiments comparing both algorithms with the same variable and value ordering heuristics show that there is a modest but significant gain of MFC ranging from 10% to 30% consistency checks.

From a practical point of view, it is mandatory to consider that FC is generally used with dynamic variable ordering heuristics based on future domain size. The efficiency of combining FC with domain-based variable ordering heuristics is nowadays out of the question [Haralick and Elliott, 80; Dechter and Meiri, 94; Bacchus and van Run, 95]. An important drawback of MFC is that it does not know the true size of future domains, so it cannot be naturally combined with these heuristics. This disadvantage is partially overcome in [Bacchus and Grove, 95] where the minimum remaining values heuristic is also computed in a lazy way. However, it decreases the power of MFC because it forces the computation of some consistency checks that otherwise would be delayed. With this approach, MFC seems to be slightly superior to FC, although no exhaustive experimental results have been published.

5.3 Lazy Propagation in Partial Constraint Satisfaction

5.3.1 Partial Forward Checking Redundancies

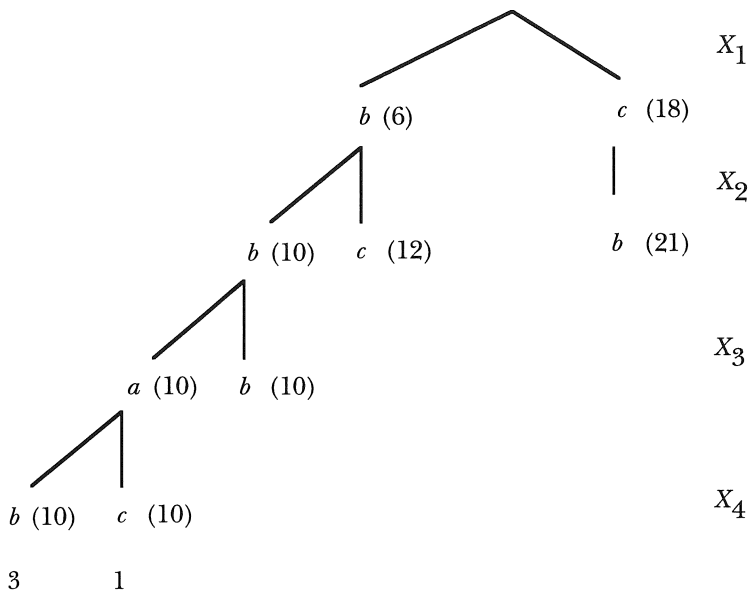
PFC, the forward checking counterpart for partial constraint satisfaction, updates ICs of future values after each new assignment and checks for feasibility right after each relevant change. There are two different places where pruning is attempted:

1. *Early pruning:* each time a value is assigned, feasibility of future values is checked during look-ahead.
2. *Late pruning:* when a value is assigned, its feasibility condition is checked again before propagating it because the upper bound may have decreased after solving a previous sibling subproblem.

Therefore, given value b of variable X_j, its feasibility condition is checked at every visited node where X_j is a future variable (early pruning). In addition, its feasibility is checked again at nodes where X_j is the current variable before attempting the assignment of b (late pruning). As in the FC case, we have observed that this greedy strategy performs consistency checks that are not strictly required. Interestingly, PFC has an additional source of redundant computation associated to its early pruning.

Example 5.3:

Consider again the previous graph coloring problem. It was shown in Example 5.1, that this problem is overconstrained. If PFC is used to find its best solution, the following tree is traversed. As in the previous examples, we indicate at each node the accumulated number of checks performed including the current propagation. In addition, at each leaf we indicate its distance which becomes the current upper bound.



For this problem, all consistency checks associated to $b \in D_3$ are redundant because PFC never makes use of ic_{3b}. The reason is that ic_{3a} is always zero, so ic_{3b} is never the minimum IC of its domain. Therefore, it never contributes to the lower bound and dead-ends can be detected without considering it. In addition, the only time $X_3 \leftarrow b$ is attempted (at node $\{X_1 \leftarrow b\ ,\ X_2 \leftarrow b\ ,\ X_3 \leftarrow b\ \}$) the current upper bound takes value 1 and the current distance is 1, too. Therefore, $b \in D_3$ unfeasibility can be detected at this node without taking into account its DAC. Moreover, this value can not be pruned at its ancestor nodes ($\{X_1 \leftarrow b\}$ or $\{X_1 \leftarrow b,\ X_2 \leftarrow b\ \}$) because their current upper bound was not yet set to 1.

PFC may perform redundant consistency checks when it tries to prune a value whose assignment is never attempted because the algorithm always backtracks before its variable is selected. What follows is a generalization to PFC of Observation 5.1.

Observation 5.3:

Let S be a search state where X_i is the current variable, a is its current value and b is a feasible value of a future variable X_j. Assume that PFC performs look-ahead and no dead-end is detected. Consequently, all consistency checks between the current assignment and future values are performed. In particular, the consistency of $(a,b) \in R_{ij}$ is checked. Suppose that PFC proceeds its search below S in the tree and finally backtracks to S satisfying the two following conditions:

- X_j has never become the current variable,

- for all detected dead-ends, the domain of X_j had at least one feasible value whose IC was lower than the IC of b,

in this situation, the consistency check $(a,b) \in R_{ij}$ performed in S and all subsequent checks associated with value b in successor nodes of S could have been avoided because the algorithm did not make any use of them.

Because of its branch and bound structure, PFC has two bounds associated with each node that change monotonically during search. The lower bound grows with the tree level and the upper bound decreases with the number of visited leaves. The algorithm evaluates the pruning condition of future values using its current lower and upper bounds. Therefore, depending on the node where pruning is attempted, a different amount of inconsistencies with respect to past variables is required. PFC early pruning is not the most economic strategy for future value pruning in terms of consistency checks.

Observation 5.4:

Let S be a search state (with **P** and **F** past and future variables) where X_i is a future variable and a is one of its feasible values with an IC greater than the minimum of its domain. Assume that a can be pruned at S, that is,

$$distance(S) + ic_{ia}(S) + \sum_{j \in \mathbf{F}\text{-}i} min_b\{ic_{jb}(S)\} \geq upper\_bound(S)$$

If PFC did not prune value a at S, PFC would reach a successor state S' (with **P'** and **F'** past and future variables), where X_i is the current variable and a the value to instantiate with. At that point, PFC tests the pruning condition for a, which obviously holds,

$$distance(S') + ic_{ia}(S') + \sum_{j \in \mathbf{F'}} min_b\{ic_{jb}(S')\} \geq upper\_bound(S')$$

where, for our convenience, we do not include ic_{ia} in the current distance. We can make two independent analyses comparing pruning conditions at S and S':

1. Regarding lower bounds for a at S and S', it is easy to see that,

$$distance(S) + \sum_{j \in \mathbf{F}\text{-}i} min_b\{ic_{jb}(S)\} \leq distance(S') + \sum_{j \in \mathbf{F'}} min_b\{ic_{jb}(S')\}$$

that is, the contribution of every variable but X_i to lower bound of S' is greater than or equal to its contribution to lower bound of S. Therefore, the required contribution from ic_{ia} to satisfy the pruning condition of a at S' is lower than or equal to the same contribution at S. So, pruning a at S' may require to update ic_{ia} against less past variables than pruning at S, which saves checks.

2. Regarding upper bounds, S' is visited after S, so there may have been an upper bound decrement from S to S'. Then,

$$upper\_bound(S') \leq upper\_bound(S)$$

Assuming that distance and sum of minimum inconsistency counts are always computed, pruning at S' may require an ic_{ia} lower than or equal to the ic_{ia} required at S. Therefore, pruning a at S' may require to update ic_{ia} against less past variables than when pruning at S, which saves checks.

Analyses 1 and 2 are independent and complementary. Considering both analyses simultaneously magnifys their effects in consistency checks saving. In conclusion, PFC future value pruning does not always use the better bounds to minimize the number of checks and it may perform more checks that required.

In general, as one delays pruning, better bounds are available. Thus, the best bounds are at the latest point where a value can be pruned. From this analysis, we observe that attempting the early pruning of a value b is a source of redundant consistency checks. On the other hand, late pruning lacks these inefficiencies because it has the best possible bounds and it is only attempted if the variable to which b belongs is finally selected.

5.3.2 Partial Lazy Forward Checking

We have described the inefficiency of PFC caused by its early pruning strategy. In order to diminish it, we have developed an algorithm that delays pruning and IC updating until they are strictly necessary. This new algorithm, called *partial lazy forward checking* (PLFC), is a generalization of MFC to MAX-CSP. These are its two main features:

1.- Lazy look-ahead:

After each assignment, PFC checks all future values updating their IC and pruning those values that have become unfeasible. If with this process some domain becomes empty, then a dead-end is detected and the algorithm backtracks. However, PLFC exploits that to correctly detect empty domains it is not necessary to check every future value. It is enough to perform look-ahead on a future domain until the value with minimum IC is found. If that value satisfies the pruning condition, all other values also satisfy it, and the domain becomes empty. Otherwise, the value with minimum IC remains unpruned so the domain is not empty. Then, PLFC look-ahead stops testing this domain and continues on another future domain. Consequently, PLFC does not prune future values, except for the one with minimum IC in its domain. PLFC performs the minimum amount of checks to correctly compute the lower bound. The rest of checks are delayed until they are strictly necessary.

2.- Late pruning:

To prevent inefficiencies associated with early pruning, PLFC only prunes values of the current variable, right before proceeding to their look-ahead. It may seem that not performing early pruning can cost consistency checks. However, this is not the case. PLFC only updates those ICs needed to detect the least inconsistent value of each variable. It only requires updating the rest of values, at most, up to the point where their IC surpasses the minimum by one. After that, no more consistency checks are needed for any value independently of whether they are feasible or unfeasible.

PLFC appears in Figure 5.2. For simplicity reasons, lexicographical variable ordering is assumed. Domains, inconsistency counts and the *cons_level* structure are global variables. At each search state, PLFC selects the current variable and considers the assignment of all its feasible values. When value a is assigned to X_i, it performs a two-step procedure: first, it *catches up* delayed consistency checks of a with respect to past variables (line 8). Second, if value a is feasible, the lazy look-ahead computes the minimal amount of checks to update the lower bound (line 9).

When computing consistency with respect to past variables, some consistency checks may have been previously performed as part of the lazy look-ahead when the current variable was still a future variable. In order to avoid their repetition, we keep a table, *cons_level*, that records up to what level the current IC counts are updated. If b is a value of a future variable X_j, whose current ic_{jb} takes value v, then $cons\_level_{jb}=u$ means that consistency of value b has been already checked with the first u past variables and it has v inconsistencies with them. Procedure *lazy_look_back*(*dist, i, a*) updates the number of inconsistencies of $X_i \leftarrow a$ with respect to past variables (which is already computed to variables up to its *cons_level* count). Each time that ic_{ia} is incremented, feasibility is tested (line 21) and the update only continues if it succeeds. If the final test succeeds, then $cons\_level_{ia}$ takes value i-1 and the function returns *true*. If during the update, the feasibility test fails after detecting the inconsistency to a past variable X_w, then $cons\_level_{ia}$ takes value w and the function returns *false*.

Function *lazy_look_ahead*(*dist, i, a*) performs the lazy upper bound updating after $X_i \leftarrow a$. It iterates on the set of future variables, updating the IC of the value having the lowest inconsistency with past variables. To do that, it uses the function *update_value_min_IC*() which updates IC of individual variable domains. After each call, the dead-end condition is checked (line 28) so the updating stops as soon as a the dead-end is detected. This function returns *true* if no dead-end is detected, *false* otherwise.

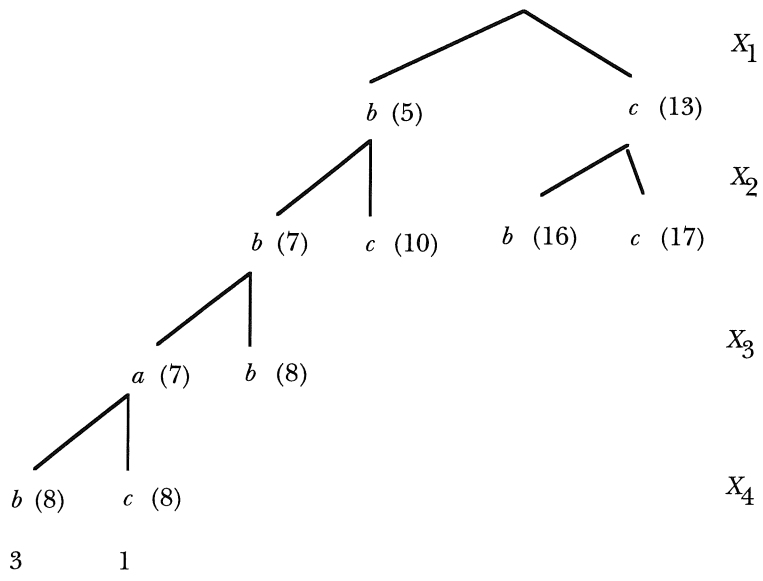
Procedure *update_value_min_IC*(j,i) updates the IC of the value in X_j having the least inconsistencies with past variables, including the current assignment of X_i. Before the call, only one value b having the minimum IC before the current assignment is guaranteed to be updated up to the i-1

variable. The rest of values have their IC only updated as necessary to know that they have more inconsistencies with past variables than b (namely, up to the point where their IC is at least equal to the IC of b). The procedure updates ICs in the following way. The inconsistency of b with the current assignment is checked. If they are consistent, we know that b still is the value with the fewest inconsistencies so the procedure stops. Otherwise, its IC is incremented and the algorithm selects a new value having a lower IC than b, and updates its IC until it is increased, or up to the current assignment. If its IC is not incremented the procedure can stop. Otherwise, it selects another value having a lower IC and repeats the process.

Finally, like in the MFC case, we provide the domain restoration code. Procedure *restore*(i, a) removes all contributions to ICs of future domains that was caused by their inconsistency with the current assignment of X_i. It also updates *cons_level* because the current assignment to X_i is going to be changed and all future values whose consistency was checked up to variable X_i are no longer updated. For clarity purposes, our implementation performs consistency checks to retrieve information. However, the same effect —but in a consistency check free manner— can be obtained by keeping a list of increased ICs associated to each past variable. In the following we assume the consistency check free implementation.

Example 5.4:

If the running example coloring problem is solved with PLFC, the following search tree is traversed,



which saves 4 consistency checks with respect to PFC. The following tables shows inconsistency counts and *cons_level* values at node $\{X_1 \leftarrow c\}$,

IC	X_2	X_3	X_4
a		0	
b	0	0	0
c	0		0

cons_level	X_2	X_3	X_4
a		1	
b	1	0	1
c	0		0

Observe that each domain has at least one IC updated (with *cons_level* taking value 1). This updated IC is guaranteed to have the minimum IC of its domain (in this case having an updated IC taking value 0 guarantees its minimality).

5.3.3 Theoretical Analysis of PLFC

In this section, we compare PLFC with PFC in terms of consistency checks. Our main result shows that PLFC never performs more consistency checks than PFC. Its proof requires the use of the following observation.

Observation 5.5:

A variable and value ordering define a CSP search tree. Each node, independently of any algorithm, has its associated upper bound, its distance and its future value inconsistency counts before and after propagating the current assignment. Considering PFC (and PLFC) lower bound, we can divide nodes into three groups: (*i*) nodes whose current assignment is not feasible before propagating it, (*ii*) nodes whose current assignment is feasible before propagating it, but unfeasible after propagating it, and (*iii*) nodes whose current assignment is feasible after propagating it. From the algorithm description we know that:

- if a node is of type (*ii*) or (*iii*), then both PFC and PLFC visit it,
- if PFC or PLFC visit a node, then its parent is of type (*iii*).

```
procedure PLFC (i, dist, Assg)
1       if (i=n+1) then
2               UB:= dist
3               Best_sol:= Assg
4       else
5               for all a ∈D_i do
6                       NAssg:= Assg ∪ {X_i←a}
7                       ndist:= dist + ic_ia
8                       if (lazy_look_back(dist, i, a, Assg)) then
9                               if (lazy_look_ahead(dist, i, a, NAssg)) then
10                                      PLFC(i+1, ndist, NAssg)
11                              endif
12                              restore(i, a)
13                      endif
14              endfor
15      endif
endprocedure

function lazy_look_back (dist, i, a, Assg) returns boolean
16      for j:= cons_level_ia+1 to i-1 do
17              cons_level_ia:= j
18              (X_j ←v^j) := consult(Assg,j)
19              if (inconsistent(X_i←a, X_j ←v^j)) then
20                      ic_ia:=ic_ia+1
21                      if (dist + ic_ia + Σ_{k=i+1}^{n} min_v{ic_kv} ≥ UB)) then
22                              return (false) endif
23              endif
24      endfor
25      return (true)
endfunction

function lazy_look_ahead(dist, i, a, NAssg) returns boolean
26      for j:=i+1 to n do
27              update_value_min_IC(j, i, NAssg)
28              if (dist + ic_ia + Σ_{k=i+1}^{n} min_v{ic_kv} ≥ UB)) then return (false) endif
29      endfor
30      return (true)
endfunction
```

Figure 5.2: Partial lazy forward checking (PLFC), assuming a lexicographical variable ordering.

```
procedure update_value_min_IC (j, i, NAssg)
31      b:= value_with_min_IC(D_j)
32      stop:=false
33      while (not stop) do
34              k:= cons_level_jb
35              if (k = i) then stop:=true
36              else
37                      k:=k+1
38                      cons_level_jb:=k
39                      (X_j ←v^j) := consult(Assg,j)
40                      if (inconsistent(X_j←b, X_j ←v^j) then ic_jb:= ic_jb +1 endif
41                      b:= value_with_min_IC(D_j)
42              endif
43      endwhile
endprocedure

procedure restore(i, a)
44      for j:=i+1 to n do for b:=1 to m  do
45              if (cons_level_jb=i) then
46                if (inconsistent(X_j←b, X_i ←a) then ic_jb:=ic_jb-1 endif
47                cons_level_jb:=i-1
48              endif
49      endfor
endprocedure
```

Figure 5.2 (cont.): Partial lazy forward checking (PLFC).

Theorem 5.2:

For any CSP, assuming the same variable selection and value selection orders, the number of consistency checks performed by PLFC is lower than or equal to the number of consistency checks performed by PFC.

Proof:

The proof of this theorem has a similar structure to proof of theorem (5.1). Consider an arbitrary overconstrained CSP solved by both PFC and PLFC. Let *Checks*(PFC)=$\{ch_1, ch_2,...,ch_r\}$ and *Checks*(PLFC)=$\{ch'_1, ch'_2,...,ch'_{r'}\}$ the set of checks that FC and MFC perform when solving the problem, respectively. Each check is defined by a tuple (X_p, v, X_q, w, S) which means that the test was $(v,w) \in R_{pq}$ and it occurred at node S.

As in the total constraint satisfaction case, the proof has the following steps: First, we show that there is an application from *Checks*(PLFC), to *Checks*(PFC). Second, we show that this application is injective. Third, we show that this application is not exhaustive. Thereafter, the proof is complete. For the sake of clarity and without loss of generality, we assume that variables and values are always selected in lexicographical order.

1. *Definition of the application:* Consider an arbitrary node, S, being visited by PLFC, where $X_i \leftarrow a$ is being attempted and an arbitrary consistency check, $(v,w) \in R_{pq}$, is performed ($p<q$). There are two possible places where this constraint check can be performed:
   - During the lazy look-back: in this case, $X_p \leftarrow v$ is an assignment made to a past variable, q is equal to i and w is equal to a. This check is defined by (X_p, v, X_i, a, S) and corresponds to a unique consistency check (X_p, v, X_i, a, S') performed by PFC. S' is the ancestor of S where X_p is the current variable. We know from observation 5.5 that PFC visits S' because all ancestors of visited nodes are of type (*iii*) and nodes of type (*iii*) are always visited by PFC. We know that PFC performs that check because it is required by PLFC to check the feasibility of a in S. Consequently, the current value cannot be pruned by PFC at S' where worse bound are available.
   - During the lazy look-ahead: in this case, S is a node of type (*ii*) or (*iii*). There are two additional possibilities.
     - It is a forward check: then p is equal to i, v is equal to a, q is a future value and w is one of its feasible values. In this case this checks is (X_i, a, X_q, w, S) and corresponds to a unique check (X_i, a, X_q, w, S) performed by PFC during its look ahead at node S. We know that PFC visits S because PFC visits all nodes of type (*ii*) and (*iii*). We know that PFC performs that check because, due to its lazy strategy, the forward checks that PLFC performs are a subset of the forward checks that PFC performs at the same node.
     - It is a backward check (performed during the lazy update of the lower bound): then $X_p \leftarrow v$ is an assignment made to a past variable, q is a future value and w is one of its values. In this case this check is (X_p, v, X_q, w, S) and corresponds to a unique check performed (X_p, v, X_q, w, S') by PFC. S' is the ancestor of S where X_p is the current variable. We know that PFC visits S' and performs that check for the same reasons of the first case.
2. *Proof of injectivity:* equal to Theorem 5.1.
3. *Proof of non exhaustivity:* Consider the situation described in Observation 5.3. It is easy to see that those redundant consistency checks performed by PFC are not performed by PLFC. Therefore, the application is not exhaustive.

5.4 Experimental Results

Nowadays, the best non-lazy algorithms for MAX-CSP are PFC with DAC information (Chapter 4) and PFC with RDS [Verfaillie et *al.*, 96]. PLFC can be easily combined with both algorithms. We have implemented lazy and non-lazy basic versions of these algorithms. Our lazy implementations only differ from PLFC in the lower bound that they use which affects lines 21 and 28 of Figure 5.2. For a fair comparison, our non lazy implementations only differ from PFC in the lower bound which affects lines 11, 22 and 26 of Figure 2.8. All implementations use min-heaps to keep domains. Heaps are appropriated data structures for domain keeping because they require constant time to access the value with minimum IC (or IC+DAC) and logarithmic time to update them when some IC is changed. They have been shown to be more efficient than lists in both PLFC and PFC based implementations.

We have evaluated the validity of our approach using random overconstrained CSP. On this model, we have performed the following experiments:

1. PLFC-DAC[1] vs. PFC-DAC on <10,10,p_1,p_2> and <15,5,p_1,p_2> classes, both using *forward degree* combined with *backward degree* (FC-BD) as static variable ordering [Larrosa and Meseguer, 96] and selecting values by increasing IC+DAC.
2. PLFC-RDS vs. PFC-RDS on <10,10,p_1,p_2> and <15,5,p_1,p_2> classes, both using FC-BD and selecting values by increasing IC.
3. PLFC-DAC vs. PFC-DAC on <10,m,4/9,9/10>, both using FC-BD, and selecting values by increasing IC+DAC.

Each data point is averaged over 100 random instances. It is not suitable to measure visited nodes when comparing lazy and non-lazy algorithms. The reason is that with the lazy approach algorithms visit more nodes than with the non-lazy. However, in these extra nodes lazy algorithms only catch up delayed work that non-lazy algorithms have already done. For this reason, in these experiments we only report consistency checks and CPU time.

Experiments (1) and (2) evaluate the performance of PLFC with DAC and with RDS against the corresponding non-lazy counterparts. Results of (1) for selected connectivities appear in Figures 5.3-5.10. Considering checks, PLFC-DAC improves PFC-DAC from 25% to more than 100%. Only for maximum tightness, PLFC-DAC and PFC-DAC perform the same number of checks. Considering CPU time, PLFC-DAC improves PFC-DAC from 25% to more than 130%. It only performs slightly worse than PFC-DAC for problems with maximum tightness, those without check improvement. Results of (2) using RDS appear in Figures 5.11-18.

[1]Both PLFC-DAC and PFC-DAC include the lower bound improvement described in Section 4.4.

Considering checks, PLFC-RDS improves PFC-RDS from 25% to more than 65%. Only for maximum tightness, PLFC-RDS and PFC-RDS perform the same number of checks. Considering CPU time, PLFC-RDS improvement over PFC-RDS is similar to the check improvement, although PLFC-RDS performs slightly worse than PFC-RDS only for problems with maximum tightness.

From experiments (1) and (2) we conclude that the lazy approach provides significant gains both regarding number of checks and CPU time. It seems to be more advantageous when combined with the DAC approach described in Chapter 4 than with the RDS approach. It may seem surprising that gains in terms of time are sometimes larger than in terms of consistency checks. One possible explanation is that non-lazy algorithms perform more IC propagation toward future values. With our implementation, heaps have to be ordered after each IC modification. Therefore, non lazy algorithms may have a larger overhead.

Experiment (3) evaluate the ratio gain of PLFC-DAC with varying domain size m. Its results appear in Figure 5.19. As it could be expected, PLFC-DAC ratio gain increases with m. Problems with large m give more opportunities to PLFC-DAC to save checks because the amount of non updated values at each domain increases.

5.5 Conclusions and Future Work

In this Chapter we have shown that lazy evaluation techniques can be successfully applied to partial constraint satisfaction. We have extended previous work on lazy evaluation in the total constraint satisfaction context to partial constraint satisfaction. This generalization is especially appropriate because PFC has more situations than FC where it performs more checks than really needed. Our algorithm optimizes the work required to compute the dead-end condition. In addition to save effort during look-ahead, it improves the way PFC prunes future values because PLFC uses better bounds than PFC.

Interestingly, PLFC can be directly combined with DAC and RDS which are the best existing algorithms for MAX-CSP. Since our experiments show that the lazy approach provides significant computational gains to both of them, their lazy implementation is probably the most efficient for MAX-CSP. However, in our experiments we used a basic version of DAC-based algorithm. A lazy implementation of the more sophisticated DAC algorithms (PLFC-GDAC, PLFC-RDAC and PLFC-MRDAC) remains as future work.

Lazy evaluation appears to be a very fruitful approach in the context of algorithms for constraint satisfaction and optimization in order to perform the minimum amount of work at each algorithmic step. It is expected that this technique can be successfully applied to other constraint

satisfaction algorithms. We are refering, for instance, about lazy versions of algorithms that maintain arc-consistency in the total constraint satisfaction context and lazy algorithms that maintain DAC in the partial constraint satisfaction context. The exploration of these ideas is left as future work.

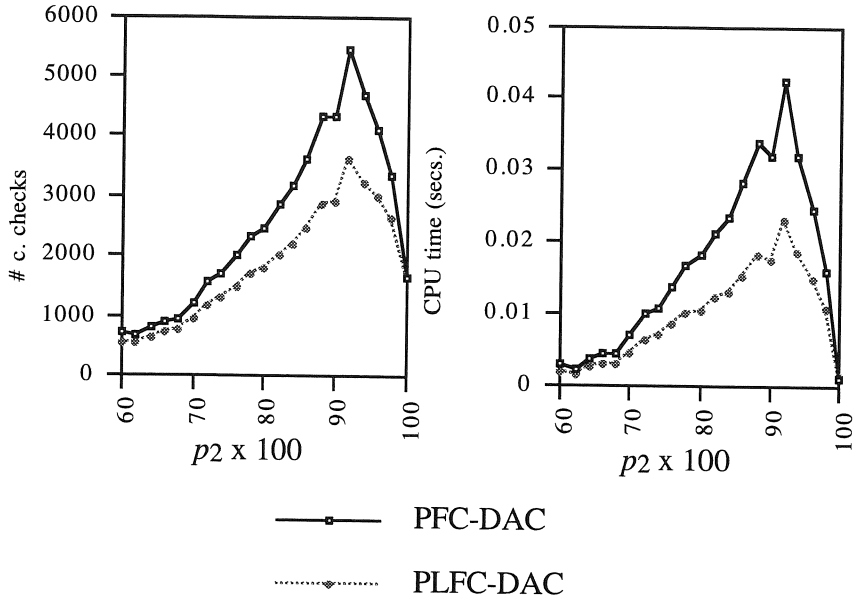


Figure 5.3: PFC-DAC vs. PLFC-DAC on the <10,10,15/45,p_2> class.

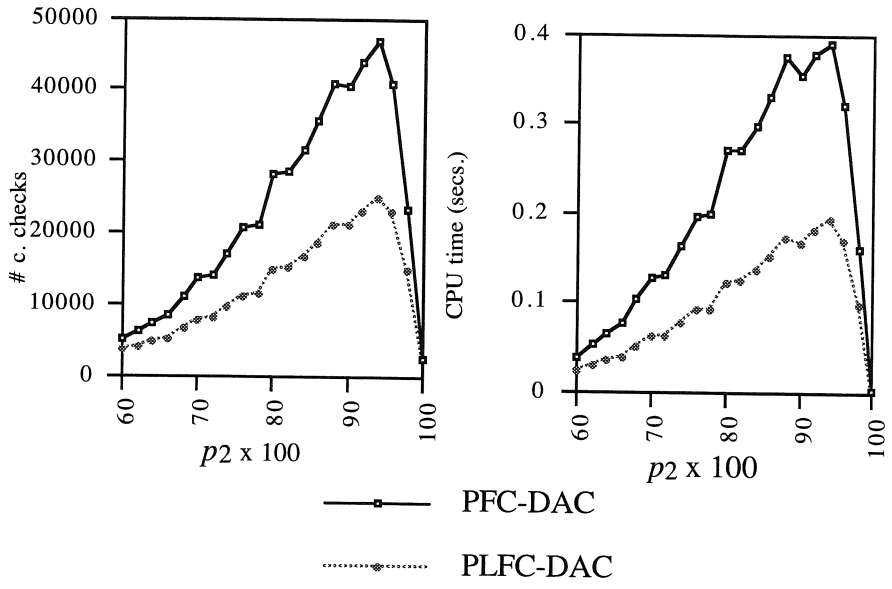


Figure 5.4: PFC-DAC vs. PLFC-DAC on the <10,10,25/45,p_2> class.

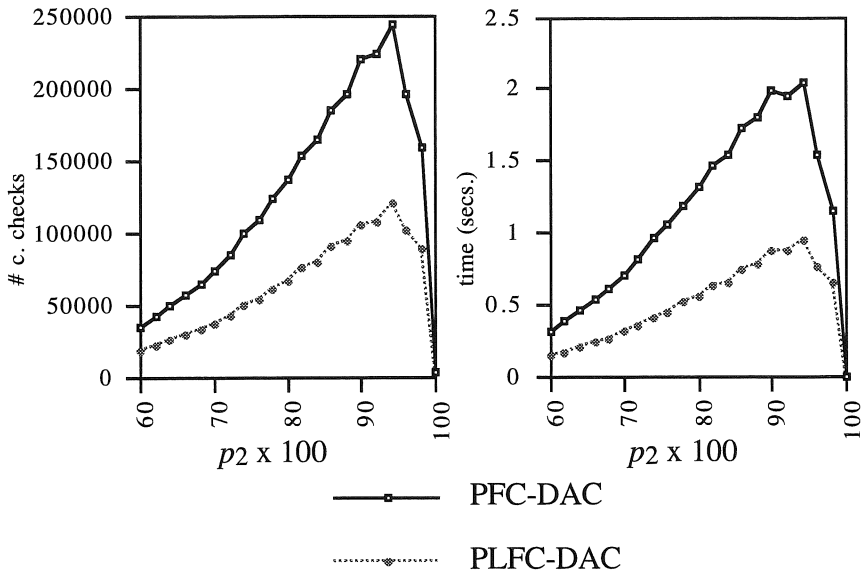


Figure 5.5: PFC-DAC vs. PLFC-DAC on the <10,10,35/45,p_2> class.

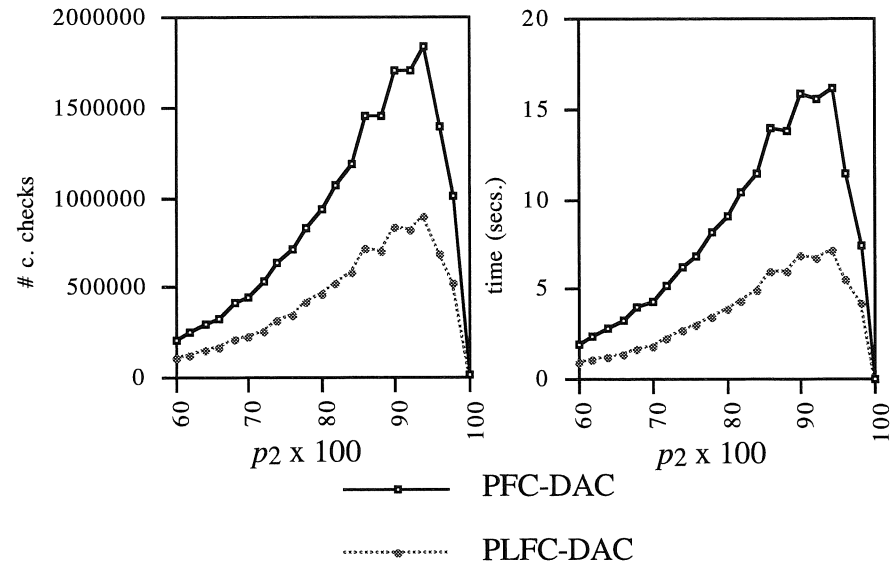


Figure 5.6: PFC-DAC vs. PLFC-DAC on the <10,10,45/45,p_2> class.

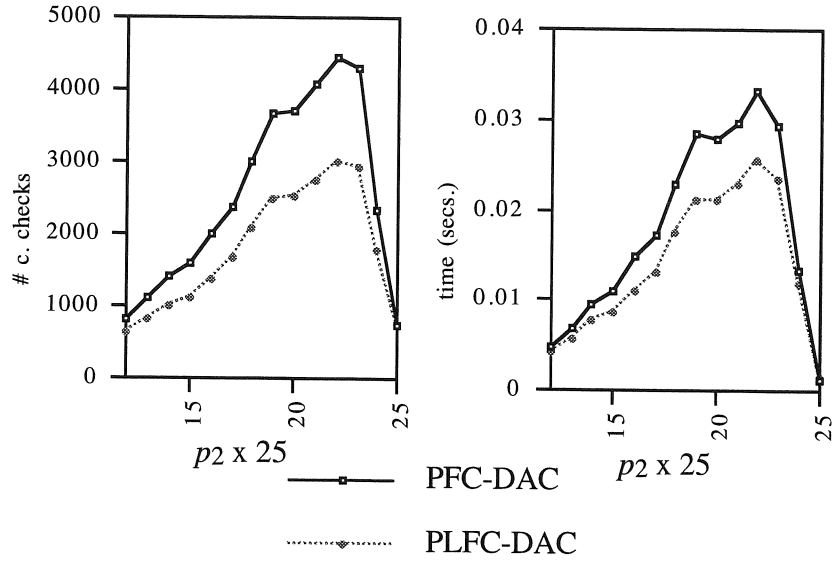


Figure 5.7: PFC-DAC vs. PLFC-DAC on the <15,5,25/105,p_2> class.

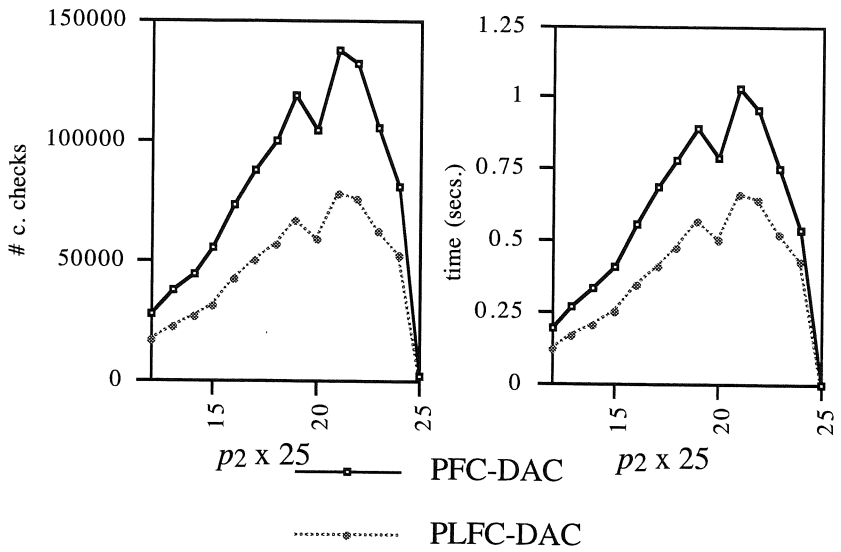


Figure 5.8: PFC-DAC vs. PLFC-DAC on the <15,5,55/105,p_2> class.

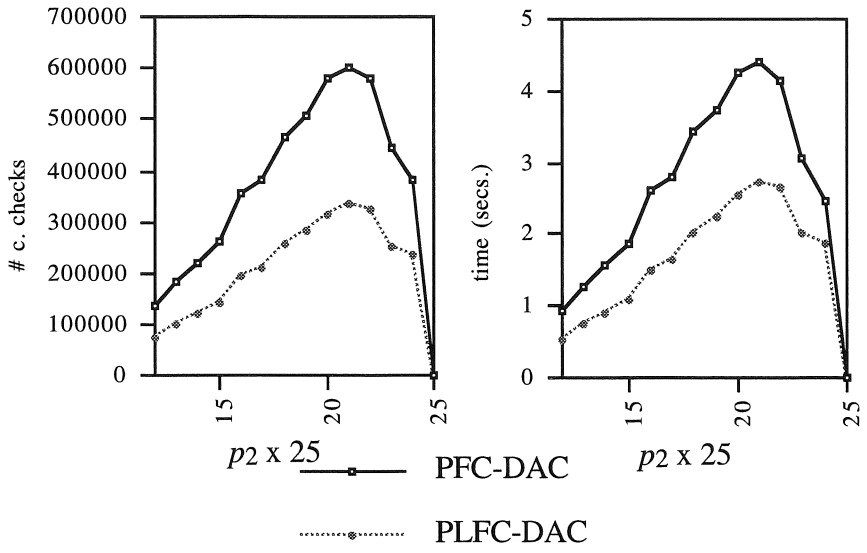


Figure 5.9: PFC-DAC vs. PLFC-DAC on the <15,5,75/105,p_2> class.

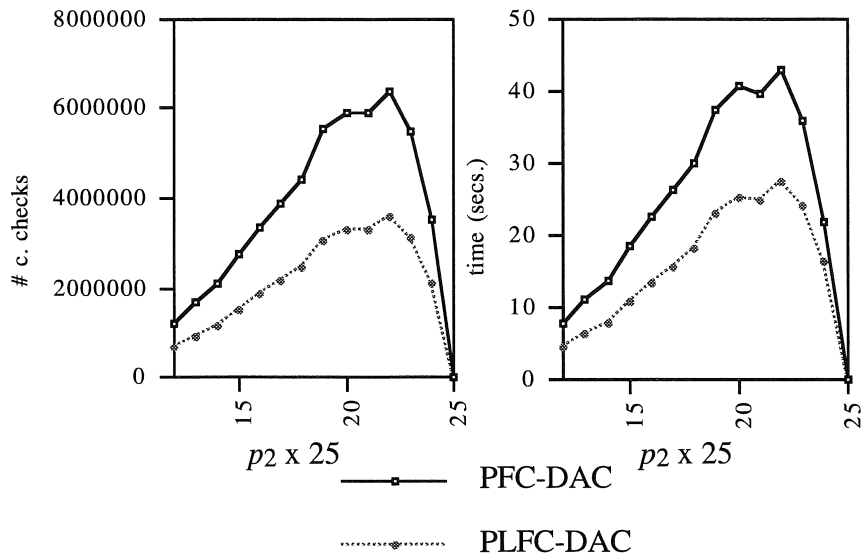


Figure 5.10: PFC-DAC vs. PLFC-DAC on the <15,5,105/105,p_2> class.

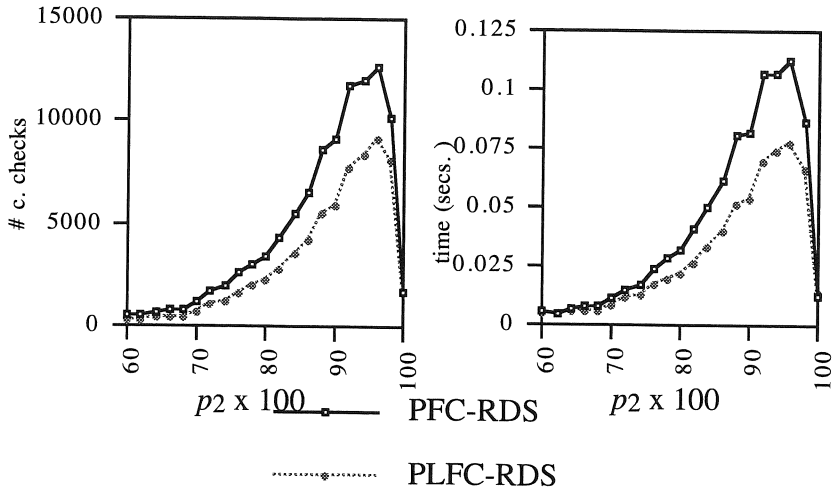


Figure 5.11: PFC-RDS vs. PLFC-RDS on the <10,10,15/45,p_2> class.

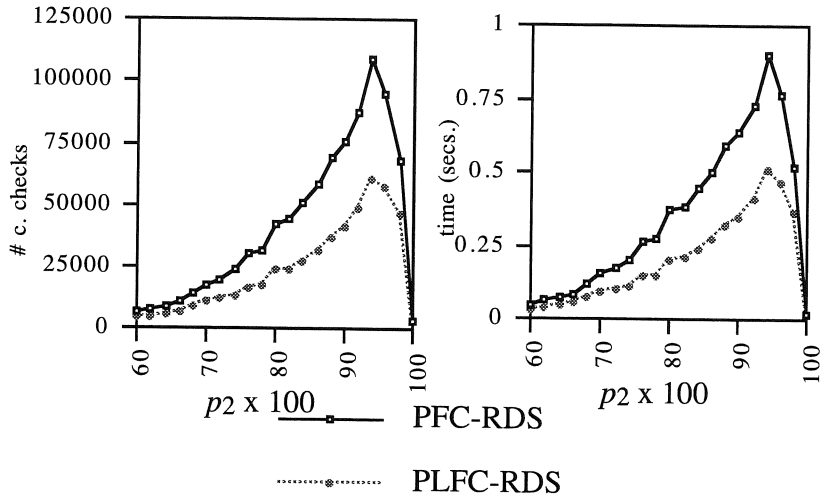


Figure 5.12: PFC-RDS vs. PLFC-RDS on the <10,10,25/45,p_2> class.

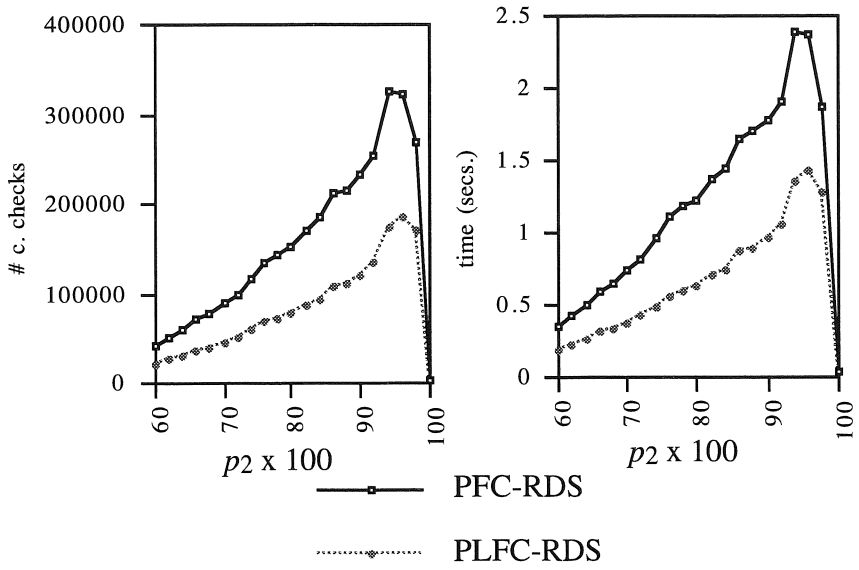


Figure 5.13: PFC-RDS vs. PLFC-RDS on the <10,10,35/45,p_2> class.

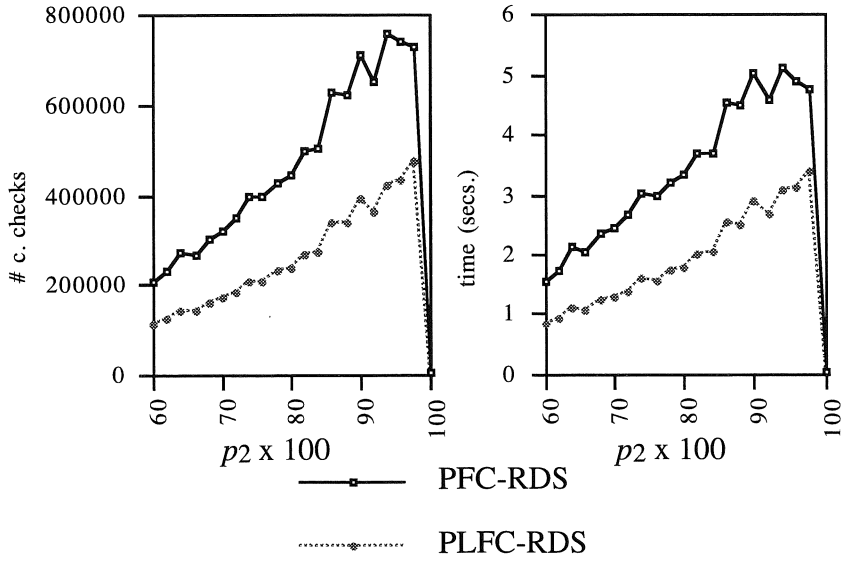


Figure 5.14: PFC-RDS vs. PLFC-RDS on the <10,10,45/45,p_2> class.

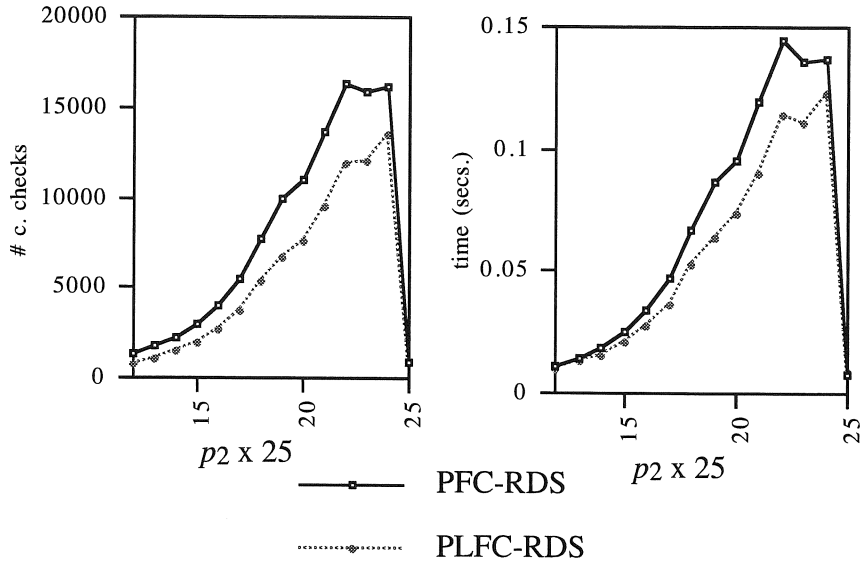


Figure 5.15: PFC-RDS vs. PLFC-RDS on the <15,5,25/105,p_2> class.

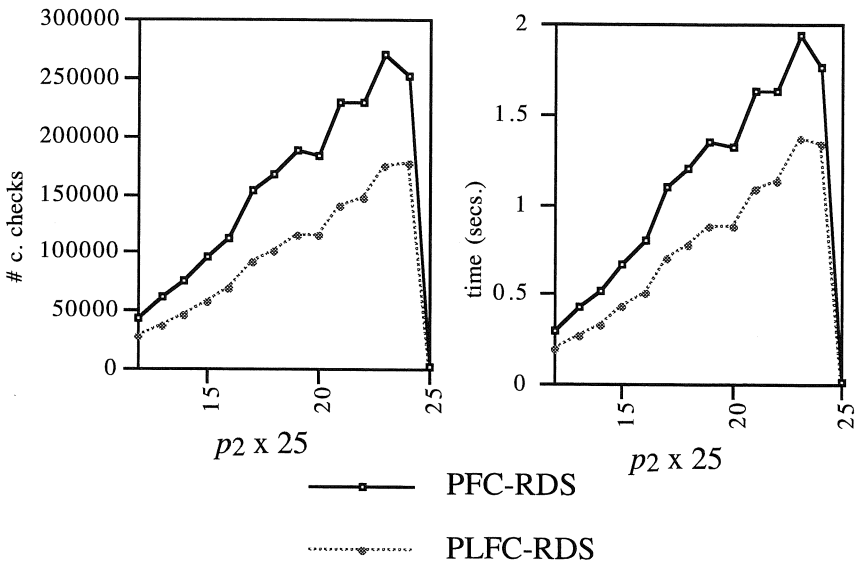


Figure 5.16: PFC-RDS vs. PLFC-RDS on the <15,5,55/105,p_2> class.

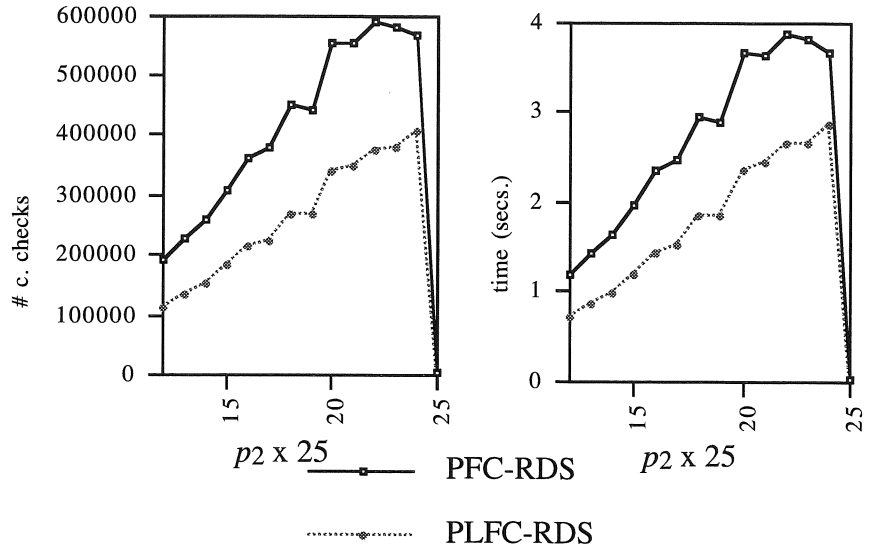


Figure 5.17: PFC-RDS vs. PLFC-RDS on the <15,5,75/105,p_2> class.

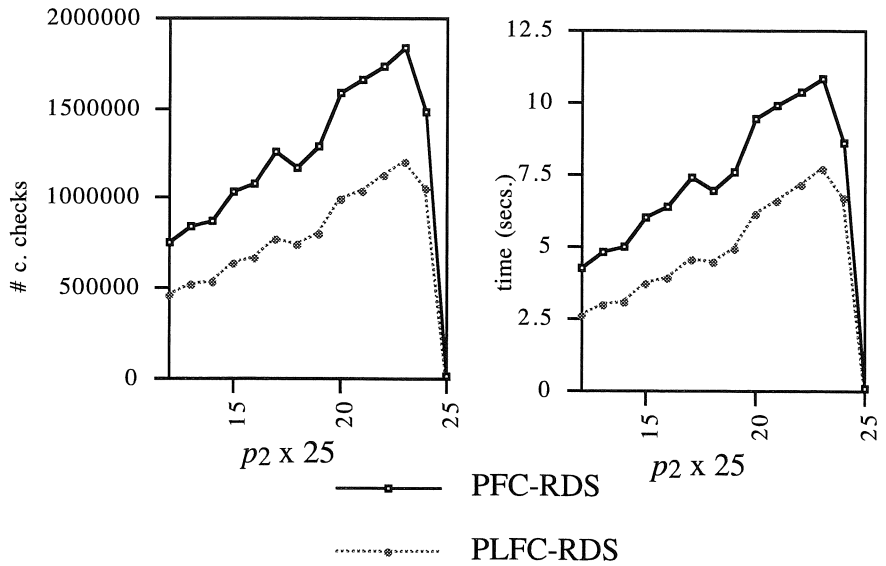


Figure 5.18: PFC-RDS vs. PLFC-RDS on the <15,5,105/105,p_2> class.

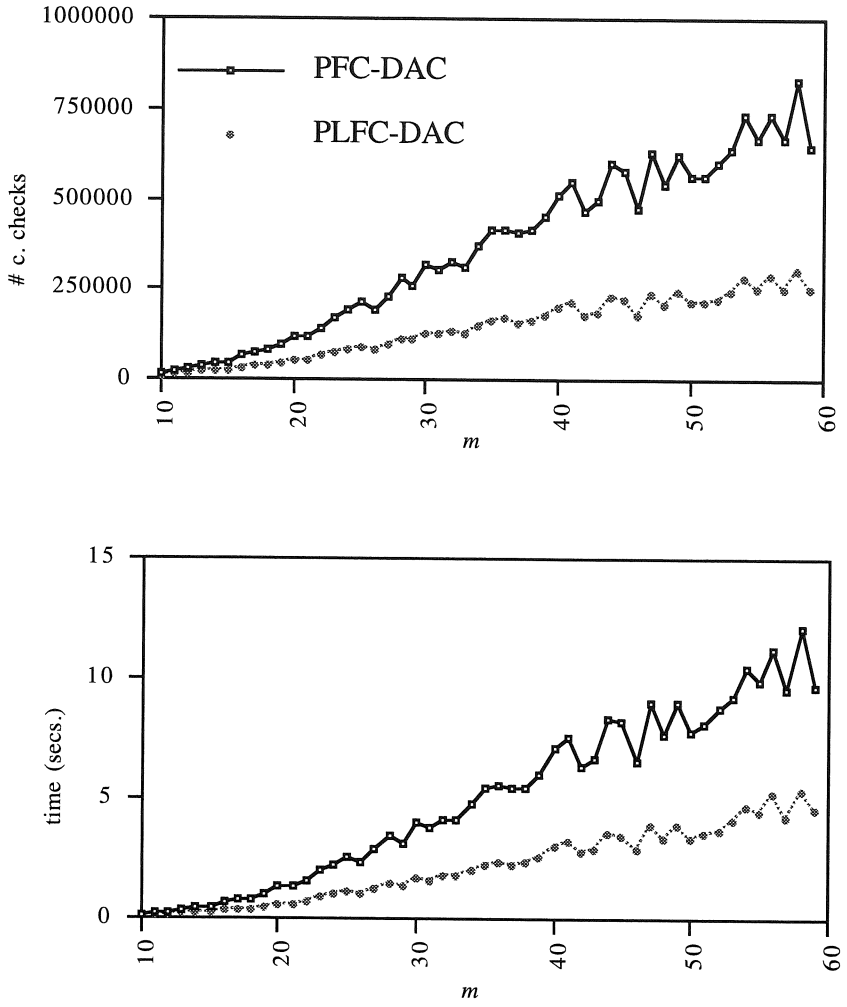


Figure 5.19: PFC-DAC vs. PLFC-DAC on the class <10,m,4/9,9/10>

Chapter 6

Support-based Heuristics

It has been known for a long time that the order in which variables and values are considered in depth-first search has a considerable impact in algorithm efficiency. For this reason, several heuristics for variable and value ordering have been developed. In this Chapter we present a new perspective of constraint satisfaction and show its usefulness for heuristic generation. Our approach is based on the analysis of the labelling problem, a formalism arising in the field of computer vision that is closely related to constraint satisfaction. From this analysis, we extract that total and partial constraint satisfaction can be seen as the global optimization of the so-called average local consistency function. We use this point of view as a source of inspiration for heuristics. As a result, we introduce a pair of heuristics for variable and value selection which base their advice in the gradient of the average local consistency function.

Comparing our heuristics with other constraint satisfaction approaches, we believe that our heuristics are more general because they have deeper foundations (they are based on gradients which have a well known topological interpretation), include variable and value orderings in the same framework and can be applied to both total and partial constraint satisfaction

The structure of this Chapter is the following. Section 6.1 is introductory. In Section 6.2, we define the labelling problem and show its relation to constraint satisfaction. In Section 6.3, we show how the concept of support (taken from the labelling problem) can be used to generate heuristics. In Section 6.4, we present a computationally more efficient version of the heuristics. In Section 6.5, we compare our approach with other heuristics. In Section 6.6, we present some experimental results. Finally, in Section 6.7, we give the conclusions of the Chapter and suggest some further work.

6.1 Introduction

The order in which variables and values are selected in depth-first algorithm defines the tree that is traversed. It has been known for a long time that these orders have a profound impact in search efficiency. The order in which values are assigned defines the location of solutions. For example, consider a problem with three values $\{a, b, c\}$ per variable such that its only solution is $\{X_1 \leftarrow a, X_2 \leftarrow a, \ldots, X_n \leftarrow a\}$. It is clear that the order in which values are assigned will affect the cost of search. On the one hand, if values are considered in lexicographical order, even the most naive algorithm will efficiently find the solution because it is located at the leftmost leaf, and the algorithm will go straight to it. On the other hand, if values are considered in counter-lexicographical order, even the most sophisticated algorithm is likely to invest substantial effort to find the solution because under this ordering the solution is located at the rightmost leaf and search will need to traverse the whole tree before finding it .

The order in which variables are selected may also affect search efficiency. To illustrate this fact consider the graph coloring problem of Figure 6.1. It is easy to see that it is unsolvable because there is no consistent assignment including the three first variables. Any algorithm selecting variables in lexicographical order will efficiently detect its unsolvability because dead-ends will be detected in the first three levels of the tree. However, algorithms selecting variables in counter-lexicographical order will require to instantiate from variable X_n to variable X_3 before being able to detect the dead-end. Then, the algorith will backtrack, change a previous assignnment and rediscover the same failure. This process will be repeated for all consistent assignments including variables $X_3, \ldots, X_n$.

In this Chapter we present new heuristics for CSP and MAX-CSP variable and value ordering. Although we assume forward checking algorithms (*i.e.*: FC and PFC), one should be aware that our approach is valid for any other look-ahead algorithm. Our heuristics are inspired in the analysis of the *labelling problem* (LP), a formalism arising in computer vision that has allowed the development of efficient algorithms for low level image processing [Rosenfeld *et. al.*, 76; Hummel and Zucker, 83; Kittler and Illingworth, 85; Torras, 89]. Roughly, LP involves the labelling of a set of units subject to a set of local preferences. Solutions are *consistent* labellings which, in a sense, maximize preferences. It is obvious that LP have much in common with CSP, since both deal with assignments that must be *consistent* with respect to some preferences.

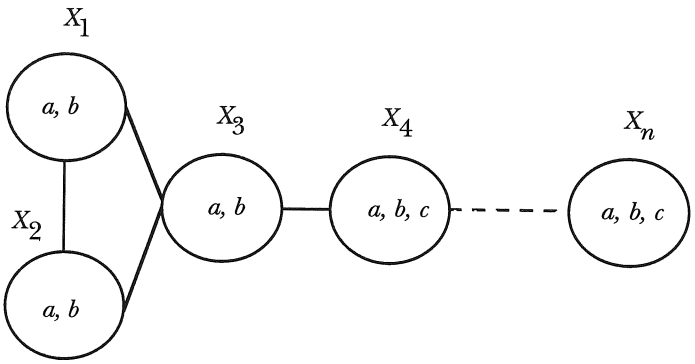


Figure 6.1: An unsolvable graph coloring problem.

LP attracted our attention because of its similarity with constraint satisfaction and the existence of effective solving methods for it. Our first objective was to explore how LP algorithms could be adapted to the constraint satisfaction context. With this purpose we carried out a theoretical analysis of similarities and differences between CSP and LP. From this analysis we extract that LP and CSP can be seen as the local and global optimization of the same function. This result explains why CSP are, in general, more difficult than LP.

Existing methods for LP solving follow a hill-climbing schema which is appropriated for local optimization but does not guarantee global optimality. If these methods are directly applied to CSP solving, search gets often trapped in local optima. The idea of CSP solving using local optimization methods has been studied before and has lead to a number of approaches to circunvent the problem of getting trapped in local optima. For instance, [Minton et *al.*, 90] presents a heuristic repair method which involves the performance of some limited amount of search in the neighbourhood of the local optima to find a way out; [Selman et *al.*, 92] propose the addition of some stochasticity to allow some random *sideway* local changes; [Morris, 93] associates a penalty cost with violated constraints at local optima, which causes a change in the function surface such that the basin disappears.

We follow a completely different approach. We show that information from the local optimization point of view can be effectively used to heuristically guide search within a systematic algorithm. In this new context, we are no longer concerned with local search issues (such as escaping from local optima) because the systematic algorithm will explore all directions until finding a solution (global optimum). In our approach, local optimization is used to devise heuristics for variable and value selection. Local optimization information is adapted to the role that these heuristics play in a systematic algorithm. The contributions presented in this Chapter are twofold:

1. From a theoretical point of view, we present a reformulation of CSP into LP which is used to clarify the common aspects between them. Basically, our results show that LP can be seen as a local optimization problem, while total and partial constraint satisfaction can be seen as a global optimization problem of the same function.
2. From a practical point of view, we import the concept of support from LP into the CSP context. In our approach, this concept (which is equivalent to a gradient) is used to devise heuristics to guide search. Interestingly, it is equally useful for both CSP and MAX-CSP and provides the first unifying view, as far as we know, for heuristic generation. Furthermore, we propose two alternatives for support computation of increasing accuracy and computing cost. Therefore, the most cost-effective for each particular situation can be chosen.

6.2 Labelling Problems and its Relation to Constraint Satisfaction

6.2.1 Labelling Problems

A LP is defined by a finite set of *units* $\{U_i\}$, a set of *labels* for each unit $\{\Lambda_i\}$, a *neighbour relation* over the units, and a *compatibility relation* over tuples of neighbour units. In our work we assume that sets of labels are discrete and finite and neighbour relations are binary. The number of units is n and, without loss of generality, we will assume a common set of indexed labels Λ for all the units, being m its cardinality.

Compatibilities are real-valued functions, r_{ij}: $\Lambda \times \Lambda \rightarrow \Re$; where $r_{ij}(a,b)$ refers to the compatibility of the simultaneous assignment of a to U_i and b to U_j. High values mean high compatibility, while low values mean low compatibility or incompatibility (the importance of these magnitudes is relative to the rest of them). We assume that compatibilities are symmetric (*i.e.*: $r_{ij}(a,b)=r_{ji}(b,a)$) and that each unit has null compatibility with itself (*i.e.*: $r_{ii}(a,b)=0$, $i=1,..,\ n;\ \forall a,b \in \Lambda$).

A *labelling* is a weighted assignment of labels to units. A weight is a real number in the interval [0, 1]. More than one label can be assigned to the same unit, provided that the sum of weights for each unit is 1. Accordingly, the set of labellings (**K**) is defined in the following way,

$$\mathbf{K} = \{V \in \Re^{n \times m} \mid 0 \leq v_{ia} \leq 1,\ i=1,...,\ n \quad a \in \Lambda;\ \sum_{a \in \Lambda} v_{ia} = 1,\ i=1,...,\ n\}$$

where v_{ia} is the weight that labelling V associates with label a of unit U_i. A labelling is *unambiguous* when it assigns one label per unit. The set of unambiguous labellings ($\mathbf{K}^*$) is defined as follows,

$$\mathbf{K}^* = \{V \in \Re^{n \times m} \mid v_{ia} \in \{0,1\},\ i=1,\ldots,n \quad a \in \Lambda;\ \sum_{a \in \Lambda} v_{ia} = 1,\ i=1,\ldots,n\}$$

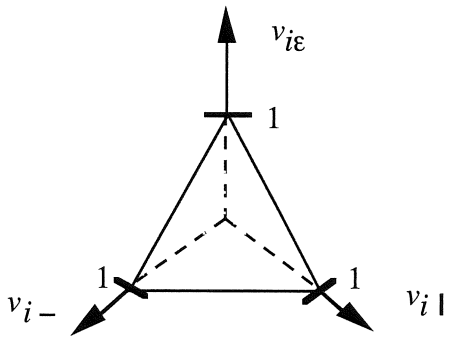
A labelling is ambiguous when it is not unambiguous. The set of ambiguous labellings is $\mathbf{K} - \mathbf{K}^*$.

In a similar way that CSP aim at finding total assignments where constraints are not violated, LP looks for labellings where units are highly compatible to their neighbours with respect to compatibility functions.

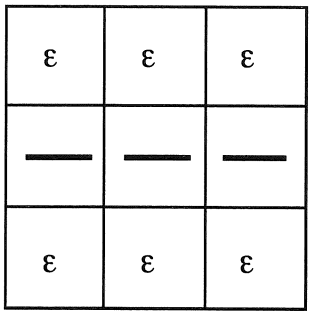
Example 6.1:

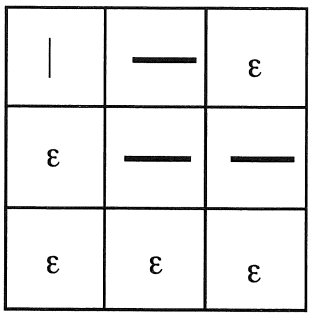
A typical labelling problem is *edge detection* in two-dimentional images [Hummel and Zucker, 83]. Consider a simplified version in which an image consist of a matrix of binary pixels (*i.e.* a pixel can be either white or black) that represents a scene formed by different segments. Each pixel may or may not belong to a segment, and segments may be of two different types: horizontal or vertical. It is assumed that segments are one pixel thick. Given an image, the task is to detect their segments by individually considering pixels. Images usually have noise, so the interpretation of an individual pixel needs not to be compatible with its neighbours. However, the overall image interpretation has to be, in general, compatible.

If we represent this situation as a LP, pixels are associated with units, the label set contains three elements: the two different segment types and the non-segment, that we denote by the following set of symbols $\Lambda=\{|,—,\varepsilon\}$. The space of possible labellings, $\mathbf{K}$, is formed by a set of bi-dimensional subspaces $\{ 0 \le v_{ia} \le 1,\ \sum_{a \in \Lambda} v_{ia} = 1\}$, one for each unit. One of these subspaces is the surface represented in the following picture.



The space of unambiguous labellings $\mathbf{K}^*$ is included in $\mathbf{K}$ and restricts labellings to the corners of each subspace.
Each pixel takes its eight surrounding pixels as its neighbours. There are different strategies regarding compatibilities. Nevertheless, they must indicate that each possible label needs to be coherent with its neighbour labels. For instance, two horizontally neighbour pixels have the following pairs of highly compatible labels: {ε,ε}, {—,—}, {ε,|} and {|,ε}. The pair of values {|,|} has a low compatibility (recall that segments are one pixel thick). The remaining possibilities have medium value compatibility because they represent corners or segment ends which are uncommon, but possible.
Consider the two following unambiguous labellings.





It is clear that the labelling at the left is more consistent with a segment-based image than the labelling at the right.

Given a labelling V, it is possible to measure the degree of compatibility of a label/unit pair with respect to one of its neighbours using the notion of *support*. The *support* that label a at unit U_i receives from unit U_j is defined as,

$$Supp(U_i, a, U_j, V) = \sum_{b \in \Lambda} v_{jb}\, r_{ij}(a,b) \qquad (6.1)$$

or, in words, the weighted sum of compatibilities that the pair (U_i, a) has with unit U_j. The *support* that label a in unit U_i receives from labelling V is the sum of all its individual supports,

$$Supp(U_i, a, V) = \sum_{j=1}^{n} Supp(U_i, a, U_j, V) \qquad (6.2)$$

If label a of unit U_i has a greater support than label b (*i.e.* $Supp(U_i,a,V) > Supp(U_i,b,V)$), it means that a is more compatible than b for V.

The *average local consistency function* of a labelling, $A(V)$, is a function globally measuring how compatible a labelling is,

$$A(V) = \sum_{i=1}^{n} \sum_{a \in \Lambda} v_{ia}\, Supp(U_i, a, V) \qquad (6.3)$$

The gradient of $A(V)$ is a function $Q(V)$: $\Re^{n \times m} \rightarrow \Re^{n \times m}$ such that

$$q_{ia}(V) = 2 \sum_{j=1}^{n} \sum_{b \in \Lambda} v_{jb}\, r_{ij}(a,b)$$

where $q_{ia}(V)$ is the gradient component associated to the v_{ia} weight. From (6.1) and (6.2), it is easy to see that gradient and supports are proportional,

$$q_{ia}(V) = 2\, Supp(U_i, a, V) \qquad (6.4)$$

LP solutions are consistent labellings. A labelling V is *consistent* provided that,

$$\sum_{a \in \Lambda} v_{ia}\, Supp(U_i, a, V) \geq \sum_{a \in \Lambda} w_{ia}\, Supp(U_i, a, V) \qquad i=1,\ldots, n \qquad \forall W \in \mathbf{K}$$

In words, a labelling V is consistent if any change of weights in one of its units does not increase the weighted support that this unit receives from the labelling. Observe that consistency in the LP sense is an optimality criterion that has to hold in n equations simultaneously. Optimality cannot be independently achieved for each equation because changing a unit weight distribution modifies supports of neighbouring units.

Relaxation techniques are a family of iterative algorithms which efficiently solve labelling problems. Starting from an initial labelling, an updating rule is repeatedly applied until a fixed point is reached. A classical relaxation algorithm which has been used as a reference point for subsequent research is due to [Rosenfeld et. *al*, 76]. They proposed the following updating rule,

$$v_{ia} := v_{ia}\, (2n + q_{ia}(V)) \;/\; \sum_{b \in \Lambda} v_{ib}\, (2n + q_{ib}(V))$$

where it is assumed that compatibilities are in the range [-1, 1]. A number of alternatives to these algorithm have been proposed in order to deal with different situations [Kittler and Illingworth, 85]. In general, updating rules increase weights that receive high supports and decrese weights that receive low support. Relaxation techniques can be seen as approximations of hill-climbing methods [Hummel and Zucker, 83].

6.2.2 Relationship between Labelling Problems and Constraint Satisfaction Problems

It is obvious that CSP and LP have much in common: both deal with assignments that must be compatible with respect to some preferences. In

this Section we analyze their similarities and differences. For this purpose, we propose a reformulation of a CSP into a LP, where each variable X_i corresponds to a unit U_i and the common domain of values D corresponds to the common set of labels Λ. For every pair of units a neighbour relation is declared where $r_{ij}(a,b)$ takes value 0 or 1 depending on whether there is a constraint disallowing that assignment in the CSP or not. The algorithm for transforming a CSP into a LP is given in Figure 6.2.

Under this transformation, each CSP variable X_i is associated with a subspace of $\mathbf{K}$ $\{0 \leq v_{ia} \leq 1, \sum_{a \in \Lambda} v_{ia} = 1\}$ and each value $a \in D_i$ has associated a weight v_{ia} of this subspace. Under this transformation there is an obvious correspondence between LP unambiguous labellings and CSP total assignments. Thus, in the sequel we will make no distinction between them.

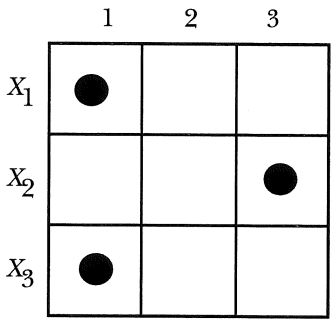
A solution for a transformed CSP is an unambiguous labeling which does not violate any constraint (*i.e.* being consistent in the CSP sense). The following example illustrates that csp-consistency[1] and lp-consistency are not equivalent concepts.

```
for every pair of variables (Xi ,Xj), i,j=1,...,n,
      for every pair of values (a,b), a,b ∈D
            if (inconsistent(Xi←a, Xj←b)) then rij(a,b) = 0
            else rij(a,b) = 1 endif
      endfor
endfor
```

Figure 6.2. Reformulation of a CSP into a LP.

Example 6.2:

Consider the 3-queens problem and the following board configuration,



[1] In the sequel, we will use the terms csp-consistency and lp-consistency to distinguish the two types of consistency.

If this CSP is transformed into a LP, the previous situation corresponds to the following unambiguous labelling $V=((1,0,0), (0,0,1), (1,0,0))$. Labelling V causes the following matrix of supports,

	1	2	3
X_1	1	1	0
X_2	0	0	2
X_3	1	1	0

Replacing these values in the definition of lp-consistency, we obtain the following equations:

$$1 \geq 1\ w_{1a} + 1\ w_{1b} + 0\ w_{1c}$$
$$2 \geq 0\ w_{2a} + 0\ w_{2b} + 2\ w_{2c}$$
$$1 \geq 1\ w_{3a} + 1\ w_{3b} + 0\ w_{3c}$$

Which obviously hold $\forall W \in \mathbf{K}$. Consequently, labelling V is lp-consistent. However, it is not csp-consistent because it violates one constraint.

Now, a natural question arises: *what is the relation between csp-consistency and lp-consistency*? In what follows we provide several results relating lp-consistency and csp-consistency with local and global maxima of the $A(V)$ function.

Theorem 6.1:[Hummel and Zucker, 1983]

Let us consider a labeling problem with symmetric binary constraints. A labeling W is lp-consistent if and only if it is a local maximum of $A(V)$.

This first result establishes the equivalence between lp-consistent labelings and local maxima of $A(V)$. These labelings can be ambiguous and therefore meaningless when considering CSP. The following theorem relates ambiguous and unambiguous lp-consistent labelings.

Theorem 6.2:[Sastry and Thathachar, 1994]

Let us consider a labeling problem with symmetric binary constraints. If there exists an ambiguous lp-consistent labeling V,

there exists an unambiguous lp-consistent labeling W such that $A(V) = A(W)$.

This theorem guarantees the existence of an unambiguous lp-consistent labeling for each ambiguous lp-consistent labeling with the same value of $A(V)$. Therefore, this theorem allow us to ignore —at least in theory— the existence of ambiguous lp-consistent labelings and to consider only unambiguous ones.

Lemma 6.1:

Let us consider a binary CSP formulated as a labeling problem and let V be an unambiguous labeling. Then, $A(V)= n(n\text{-}1)\text{-}2V_{inc}$, where V_{inc} is the number of constraint inconsistencies of V.

Proof:

It is easy to see that $n - Supp(U_i, a, V) - 1$ is exactly the number of constraint violations in V involving $X_i \leftarrow a$. After this fact, the proof is straightforward.

This lemma relates average local consistency with constraint violations. It is restricted to unambiguous labellings because ambiguity is meaningless in the CSP context. It shows that $A(V)$ is essentially counting satisfied constraints.

Theorem 6.3:

Let us consider a binary CSP formulated as a labeling problem and let W be an unambiguous labeling. W violates a minimal number of constraints if and only if W is a global maximum of $A(V)$.

Proof:

1. From left to right. Lets suppose that $A(W)$ is not a global maximum. Then, there is an labelling W' such that $A(W') > A(W)$. We can assume that W' is unambiguous because of theorem 6.2. Lets denote W_{inc} and W'_{inc} the number of constraint inconsistencies of W and W', respectively. Using Lemma 6.1, we know that,

$$n(n\text{-}1)\text{-}2W'_{inc} > n(n\text{-}1)\text{-}2W_{inc}$$

   Therefore, it is easy to see that $W'_{inc} < W_{inc}$. Then, W does not violate a minimal number of constraints.
2. From right to left. Lets suppose that W does not violate a minimal number of constraints. Then there is another unambiguous assignment W' such that violates fewer constraints than W. Lets denote W_{inc} and W'_{inc} the number of constraint inconsistencies of

W and W', respectively. It is clear that $W'_{inc} < W_{inc}$. Therefore, using Lemma 6.1 we know that $A(W') > A(W)$ what shows that W is not a global maximum.

This theorem gives a necessary and sufficient condition for the best possible solution of a CSP: it must be a global maximum of $A(V)$. The theorem is restricted to unambiguous labelings because ambiguous global maxima can exist. However, Theorem 6.2 assures the existence of unambiguous global maxima.

Corollary 6.1:

Let us consider a binary CSP formulated as a labeling problem. An unambiguous labeling V is csp-consistent if and only if $A(V) = n(n\text{-}1)$.

Proof:

This corollary is a trivial specialization of Lemma 6.1 when the total assignment satisfies all the constraints.

An implication of all these theoretical results is that we can always solve a CSP with the following procedure: reformulate the problem as a LP, compute the unambiguous global maximum of A(V) using any optimization technique and return the optimum as the best possible solution to the problem. If the average local consistency in the optimum equals $n(n\text{-}1)$, the problem is solvable and the labelling is its solution, otherwise the problem is overconstrained and the labelling is the solution of the corresponding MAX-CSP problem.

Maximazing A(V) using optimization techniques is far from an easy task. A(V) is a quadratic function generally neither convex nor concave and no efficient algorithm for its optimization is known. Nevertheless, the optimization perspective of constraint satisfaction is still useful for two reasons. On the one hand, it provides a unifying view for CSP and MAX-CSP which have been typically considered related, but different problems. On the other hand, it allows the use of optimization concepts within tree-based search. Thus, in the following Section we use this optimization perspective to generate general purpose heuristics that can be used either by CSP and MAX-CSP algorithms to heuristically guide their search.

6.3 Using Support to Guide Search

6.3.1 The Role of Heuristics in Search

In this Subsection we analyze how the order in which variables and values are selected can affect the search efficiency, and how we can use heuristic orderings to improve it. We take the number of visited nodes as the search quality measurement. Thus, good orderings produce tree traversals in which few nodes are visited. However, if deciding a good ordering requires much computation, it may not pay off. In the remaining of this Chapter we assume forward checking algorithms (*i.e.*: FC and PFC). However, one should be aware that all conclusions drawn can be applied to other look-ahead algorithms such as MAC, DAC-based PFC, PFC-RDS, etc.

We analyze separately total and partial constraint satisfaction. However, as it will become apparent at the end, the situation is very similar in both cases. Therefore, it is suitable to devise heuristics for them under a common strategy.

Total constraint satisfaction

Consider an arbitrary node where the dead-end condition does not hold. At this point, FC selects the next variable to assign and the order in which its values are attemped. Thus, these decisions transform the current subproblem into a sequence of simpler subproblems. We analyze how different decisions can affect the overall cost. There are two possible situations:

1. *The current node is in a dead-end*: In that case, each subproblem in the sequence will report failure. All of them have to be solved before reporting failure from the current node. When solving each subprolem, every path will lead to a dead-end detection. Aiming at efficiency, the objective is to detect those dead-ends as soon as possible, so search can rapidly escape from the current subtree.
   The selected variable defines the set of subproblems that are to be solved. Aiming at early dead-end detection for all of them, a good idea is to select the variable that is more likely to make the dead-end apparent. This criterion for variable selection has been often called the *fail-first* principle because it selects first those variables that are more likely to drive search to a failure.
   Value ordering is irrelevant in this case because the order in which subproblems are considered does not affect the cost of solving them.

2. *The current node is not in a dead-end*: In that case, both variable and value selection affect search. Search can stop as soon as one subproblem is successfully solved. Therefore, the most important goal is to select first a solvable subproblem because the rest of subproblems will not need to be solved.
Value ordering can achieve this objective by itself. A good idea is to select first a value that is more likely to participate in a solution subject to previous decisions. We denote this criterion for value selection the *succeed-first* principle because it selects first that value that is more likely to drive search to success.
If a perfect value ordering can be guaranteed, variable ordering becomes irrelevant because there is at least one solvable subproblem associated with a value for every unassigned variable. No matter what variable is chosen, selecting always the value associated with the solvable subproblem will lead us to the solution without backtracking. However, value orderings are far from being perfect. Therefore, the importance of variable ordering becomes apparent. A good idea for variable selection is to be pessimistic about the subsequent value selections and assume that they will make mistakes and drive search into dead-end nodes. In that context, it is meaningful to select a variable that will promote the anticipation of potential future dead-ends. This can also be done following the fail-first principle.

In general, when an algorithm visits a node it is not known whether it is in a dead-end or not. In that context, it is reasonable to devise variable and value heuristics under the following rules of thumb:

1. *Variable ordering*: follow the fail-first principle. If the current node is in a dead-end, a good choice under this criterion will produce an early dead-end detection, no matter what line of search is followed, so the algorithm will backtrack shortly. If the current node is not in a dead-end, the fail-first principle is also appropriated as a prevention for future value ordering mistakes.
2. *Value ordering*: follow the succeed-first principle. If the current node is not in a dead-end, a good choice under this criterion will maintain search out of the dead-end, so the remaining sibling subproblems will not need to be solved. If the current node is in a dead-end, all values will have to be attempted. Then, all value orderings are equally good.

If a tree search requires to visit N nodes to find a solution of a problem, only the n nodes that belong to the path to the solution are not in a dead-end. Therefore, when solving a non trivial problem, algorithms spend most of their time visiting dead-end nodes. For this reason, during most of the search value ordering is irrelevant and only causes overhead. In that context, it is clear that the fail-first principle of variable selection plays a more relevant role in search than the succeed-first principle of value ordering.

Example 6.3:

A well-known implementation of the fail-first principle for variable ordering is *minimum domains* (see Section 2.2.5). If we assume that the current node is in a dead-end, the cost of solving the current subproblem is the sum of costs of all subproblems of the selected variable. If we suppose that all possible successors from the current node are equally costly, selecting the variable having the fewest feasible values in its domain minimizes the overall cost, and minimum domains is the best heuristic under these assumptions.

Partial Constraint Satisfaction

Consider an arbitrary node where the dead-end condition does not hold. As in the previous case, PFC selects the next variable to assign and the order in which its values are attempted. Thus, these decisions transform a subproblem into a sequence of simpler subproblems. Unlike the previous case, branch and bound always solves every subproblem in the sequence. We analyze how different decisions can affect the cost of search. There are two possible situations:

1. *The current node is in a dead-end*: When solving each subprolem, every path will lead to a dead-end detection because there is no leaf improving the current upper bound. Aiming at efficiency, the objective is to detect dead-ends as soon as possible, so search can rapidly escape from the current subtree.
   The selected variable defines the set of subproblems that are to be solved. Aiming at an early dead-end detection, a good idea is to select that variable that is more likely to make the dead-end apparent. This is again the fail-first principle. In branch and bound, dead-ends occur when the lower bound becomes greater than or equal to the upper bound. In this case the upper bound remains fixed, so the fail-first principle involves selecting variables promoting high lower bounds.
   All subproblems will be solved without changing the upper bound. Then, the order in which they are considered does not affect the cost of search. Therefore, value ordering is irrelevant.
2. *The current node is not in a dead-end*: Since the current node is not in a dead-end, some of the subproblems (at least one) correspond to non dead-end nodes. When solving these subproblems, at least one leaf will be visited and the current upper bound will be decreased. Although search can improve the upper bound several times during the traversal, the only leaf that must strictly visit is that one having the best solution of the current subproblem. Aiming at an efficient traversal, the objective is to detect as soon as possible that any path not leading to the best solution is fruitless. The detection of these

dead-ends depends on both the upper and the lower bound. The evolution of the upper bound depends on the order in which subproblems are solved (value ordering) and the lower bound depends on the variable selected.

A good idea for variable ordering is to select a variable promoting high lower bounds because it will be useful for future dead-end detections. This is again to follow the fail-first principle because it aims to anticipate failure.

A good idea for value ordering is to select first that value whose subproblem has the best solution. Then, the remaining subproblems will be solved with the best possible upper bound and their dead-ends will be earlier detected. Following this idea can be also cast in the succeed-first principle, since it also involves selecting first values that are more likely to drive search to the best solution.

In general, when an algorithm visits a node it is not known whether it is in a dead-end or not. Therefore, like in the FC case, it is reasonable to devise variable and value heuristics in terms of succeed-first and fail-first:

1. *Value ordering*: follow the succeed-first principle. If the current node is not in a dead-end a good choice under this criterion will produce low upper bounds quickly, so it will help to anticipate dead-end detection in subsequent search. If the current node is in a dead-end, value ordering does not have any effect.
2. *Variable ordering*: follow the fail-first principle. Independently whether the current node is in a dead-end or not, the whole subtree has to be traversed and search will visit dead-end nodes. In that context, following the fail-first principle promotes early dead-end detection, so the algorithm will backtrack shortly.

During a search traversal, only a small polynomial number of the visited nodes can be non-dead-end nodes. The reason is that the upper bound can only be improved a polynomial number of times (the number of problem constraints) and only paths leading to an upper bound improvement are formed by non dead-end nodes. Therefore, MAX-CSP algorithms also spend most of their search visiting dead-end nodes when they solve a non trivial problem. Consequently, in MAX-CSP the fail-first principle in variable ordering also plays a more relevant role than the succeed-first principle in value ordering. Most of the time, value ordering cannot improve search and is causing overhead.

6.3.2 Support-based Heuristics

We can use local information from the optimization perspective of constraint satisfaction to implement the fail-first and the succeed-first principles. Our approach relies on a correspondence between search states and labellings. Given an arbitrary node defined by a partial assignment $\{X_j \leftarrow w^j: X_j \in \mathbf{P}\}$, its corresponding labelling V, is defined as follows,

if $X_j \in \mathbf{P}$, then	$v_{jb} = 1$,	b is w^j
	$v_{jb} = 0$,	otherwise
if $X_j \notin \mathbf{P}$, then	$v_{jb} = 1/\ \lvert Feasible(D_j)\rvert$,	b is in $Feasible(D_j)$
	$v_{jb} = 0$,	otherwise

where D_j and $Feasible(D_j)$ are the initial and the current domain of X_j, respectively. When a value is not in $Feasible(D_j)$ it means that it has been pruned. Labels corresponding to past variables are unambiguously assigned, while labels corresponding to future variables are ambiguously assigned, with a homogeneous distribution of weights among feasible values. Pruned values take weight zero.

Under this correspondence, a depth-first algorithm can be seen as a procedure which builds an unambiguous labelling as it deepens in the tree. At each search state, past variables are unambiguously labelled and future variables are ambiguously labelled. Variable selection chooses the next subspace to disambiguate and value ordering decides the order in which the possible disambiguations will be attempted.

Example 6.4:

Consider the 5-queens problem. The initial problem, when no queen is placed in the board, has the following associated labelling,

	1	2	3	4	5
X_1	1/5	1/5	1/5	1/5	1/5
X_2	1/5	1/5	1/5	1/5	1/5
X_3	1/5	1/5	1/5	1/5	1/5
X_4	1/5	1/5	1/5	1/5	1/5
X_5	1/5	1/5	1/5	1/5	1/5

that is, an homogeneous distribution of weights among feasible values. Initially, all values are feasible and all domains have size 5. Consequently, the corresponding labelling has weight 1/5 in all its components. If FC moves to a node defined by the assignment $\{X_3 \leftarrow 1\}$ and propagates its effect, the following labelling is associated with it,

	1	2	3	4	5
X_1	0	1/3	0	1/3	1/3
X_2	0	0	1/3	1/3	1/3
X_3	1	0	0	0	0
X_4	0	0	1/3	1/3	1/3
X_5	0	1/3	0	1/3	1/3

which gives weight zero to unfeasible future values and to non assigned past values. Feasible future values receive weight one divided by their domains size. Weight v_{31} takes the maximum value because it corresponds to a past assignment.

Given an arbitrary node, we can compute the support that each future value receives from the current labelling. Each support receives contributions from past and future variables. Thus, we can rewrite (6.2) in terms of past and future supports,

$$Supp(X_i, a) = Supp^{\mathbf{P}}(X_i, a) + Supp^{\mathbf{F}}(X_i, a)$$

where each contribution has the following form,

$$Supp^{\mathbf{P}}(X_i, a) = \sum_{j \in \mathbf{P}} r_{ij}(a, v^j)$$

and

$$Supp^{\mathbf{F}}(X_i, a) = \sum_{j \in \mathbf{F}} \left(\frac{1}{|Feasible(D_j)|} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) \right)$$

where we omit the V parameter because it is clear from the context. In words, each support receives for each consistent past variable a unit

contribution and for each future variable the ratio of the number of consistent values to the number of feasible values.

If supports are used with FC, feasible future values are consistent with every past variable. Consequently, the contribution from past variables to every support is equal to the number of past variables,

$$Supp^{P}(X_i, a) = |\mathbf{P}|$$

Regarding the contribution from future variables to $Supp(X_i, a)$, only those feasible values that are consistent with the considered value contribute to its support. In FC, consistent values are those values that will survive pruning if the considered value is assigned. Therefore, we can rewrite the contribution of future variables to its support as the following expresion,

$$Supp^{F}(X_i, a) = \sum_{f \in \mathbf{F}} \frac{|Feasible'(D_f)|}{|Feasible(D_f)|}$$

where $Feasible'(D_f)$ denotes the set of feasible values in D_f if $X_i \leftarrow a$ is propagated.

If supports are used with PFC, different future values may have different inconsistencies with past variables. Then, the contribution from past variables counts the number of consistencies and it can be rewritten in terms of IC,

$$Supp^{P}(X_i, a) = |\mathbf{P}| - ic_{ia}$$

The contribution from future variables to supports measures the ratio of variables whose IC will be incremented for each future variable if the considered value is assigned.

Example 6.5:

In the previous example we showed the labelling associated with the initial board configuration of the 5-queens problem. The following picture shows the support for every feasible value in that search state. The pair ($X3$, 3) receives the lowest support from this board configuration. It indicates that the assignment $X3 \leftarrow 3$ is the most inconsistent with future domains. Thus, propagating this assignment will make explicit many inconsistencies. Therefore, the resulting subproblem is likely to be close (closer than other alternatives) to a dead-end detection.

	1	2	3	4	5
X_1	12/5	12/5	12/5	12/5	12/5
X_2	12/5	10/5	10/5	10/5	12/5
X_3	12/5	10/5	8/5	10/5	12/5
X_4	12/5	10/5	10/5	10/5	12/5
X_5	12/5	12/5	12/5	12/5	12/5

We can think of support $Supp(X_i, a)$ as an indicator of how much the propagation of $X_i \leftarrow a$ will bring near the dead-end detection. In FC, a dead-end occurs when there is an empty domain. If a value has a low support, propagating its assignment reduces future domains. Thus, the resulting subproblem is more likely to be near a dead-end detection than another subproblem whose corresponding value has a higher support. Regarding PFC, a dead-end occurs when the lower bound reaches the upper bound, but the situation is similar. If a value has a low support, it is likely to have many inconsistencies with respect past and future variables, so propagating its assignment causes a new subproblem in which many new inconsistencies become apparent (inconsistencies with past variables become part of the current distance and inconsistencies with future values become IC increments). Thus, the resulting subproblem is more likely to be near the dead-end detection, too.

We mentioned in the previous Subsection that the fail-first principle involves the selection of that variable producing the apparently most immediate dead-end detection. We propose the use of future value supports to estimate how much their propagation will bring near the dead-end condition. Then, we propose the lowest support heuristic, a support-based implementation of the fail-first principle, in which we select the variable receiving the lowest support.

Definition 6.1:

> Given an arbitrary node, the *lowest support* variable ordering heuristic (LS) selects the future variable which minimizes the following expression,

$$\min_i \{ \sum_{a \in Feasible(D_i)} Supp(X_i, a) \}$$

or, what is equivalent,

$$\min_i \{ \sum_{a \in Feasible(D_i)} (\sum_{j \in \mathbf{P}} r_{ij}(a,\dot{w_j}) + \sum_{j \in \mathbf{F}} (\frac{1}{|Feasible(D_j)|} \sum_{b \in Feasible(D_j)} r_{ij}(a,b))) \}$$

The lowest support heuristic bases its decision on two aspects. On the one hand, it considers the current domain cardinality because each variable receives one support contribution for each feasible value that it has. Thus, given two variables whose values are equally supported, LS prefers the variable with minimum domain. On the other hand, LS considers the constraining behaviour of each domain value subject to the current subproblem. Thus, given two variables with the same domain cardinality, LS prefers that one whose values are less supported. Therefore, in general terms LS selects variables with few and low supported values in their domain.

The succeed-first principle for value ordering involves the selection of values producing subproblems that apparently maintain search out of dead-ends. As in the previous case, we take the support of each future value as an indicator of how much their propagation will bring near the dead-end condition. We then propose the highest support heuristic, a support-based implementation of the succeed-first principle in which we select first the value having the highest support.

Definition 6.2:

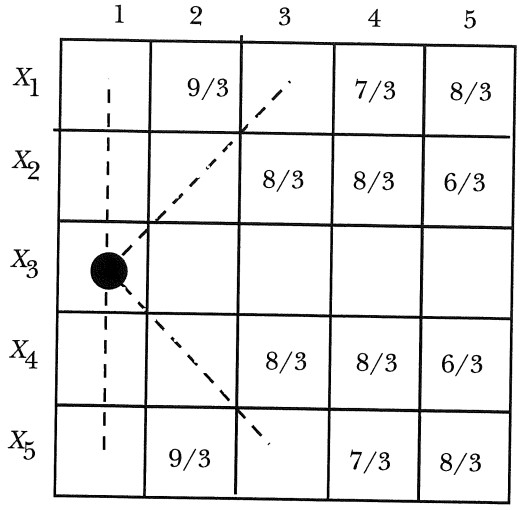
Given an arbitrary subproblem such that its current variable is X_i, the *highest support* value ordering heuristic (HS) attempts the assignment of its feasible values by decreasing support,

$$Supp(X_i, a) = \sum_{j \in \mathbf{P}} r_{ij}(a,\dot{w_j}) + \sum_{j \in \mathbf{F}} (\frac{1}{|Feasible(D_j)|} \sum_{b \in Feasible(D_j)} r_{ij}(a,b))$$

An important feature of LS and HS is that they can be used in both total and partial constraint satisfaction. If combined with FC, these heuristics take their decisions considering for each future value how constrained it is with the rest of future variables. If combined with PFC, they also consider how constrained values are with past variables.

Example 6.6:

Consider the empty assignment in the 5-queens problem. In Example 6.5 we showed its future value supports. In this situation, LS advise is to start assigning X_3 because this variable has the lowest sum of supports. Regarding value ordering, HS proposes the sequence 1, 5, 2, 4, 3 (we break ties lexicographicaly). Thus, FC would move to the node defined by $\{X_3 \leftarrow 1\}$. This search state has the following future value supports,



Therefore, LS and HS choose to continue search moving to the node defined by $\{X_3 \leftarrow 1, X_2 \leftarrow 3\}$. If this process is followed, FC finds the solution $\{X_1 \leftarrow 5, X_2 \leftarrow 3, X_3 \leftarrow 1, X_4 \leftarrow 4, X_5 \leftarrow 2\}$ without any backtracking.

6.3.3 Heuristics and Local Optimization

In this subsection we address the relation between the proposed heuristics (LS and HS) and the local optimization perspective presented in Section 6.2. Heuristics are expressed in terms of supports which are the gradients of the function $A(V)$. In this context, a natural question is: *are these heuristics implementing some kind of local optimization?*

We want to make clear that these heuristics are defined to be used inside a systematic algorithm, which considers all possible combinations of assignments, irrespective of their associated gradient. Regardless of the variable and value ordering used, a systematic traversal does not perform local optimization.

Depth-first algorithms leave room to decide the selection order of variables and values which, from the analysis of 6.3.1, should follow the fail-first and succeed-first principle, respectively. We have implemented this principles using the gradients of the function $A(V)$ as a source of local information. In this sense, we can say that our heuristics are inspired by local optimization, although the advise that they give does not have a local optimization interpretation.

Most heuristics are developed by selecting information that is believed to be relevant and combining it in an ad-hoc manner. In this context, we believe that our approach is more appropriated because we do not find how to combine information experimentally. On the contrary, we use the information given by the gradient, which has a clear foundation and a clear topological interpretation.

6.4 Incremental Support

Applying LS and HS requires the computation of supports at each visited node. The cost of computing an individual support is $O(n \cdot m)$. Consequently, the cost of computing all future value supports (required for LS) is $O(n^2 \cdot m^2)$. If a non support-based variable ordering heuristic is used, then only supports for the current variable are needed. Thus, the cost of computing them (required for HS) is $O(n \cdot m^2)$. LS and HS are clearly costly compared to other heuristics (for instance, the overhead of *minimum domain* is $O(n)$ and the overhead of sorting values by IC+DAC is $O(m \cdot \log m)$). Preliminary experiments on the n-queens problem showed that it was not cost effective to compute gradients at each node although the heuristic advise was very good (we observed a low number of visited nodes at the cost of a high number of consistency checks) [Meseguer and Larrosa, 95].

To circumvent that difficulty, we have developed an approximation of the heuristics that is much cheaper to compute. Computing the contribution of future variables to supports is what makes them costly. The reason is that this contribution depends on the current domain, so it has to be re-computed after each propagation. In our approximation, we disregard the effect that pruning has in supports. With this idea, the contribution that a variable X_k gives to the rest of variable supports does not change during search, as far as X_k remains being a future variable. To do this efficiently, we need to compute the individual contribution that every variable gives to every support at the initial problem,

$$ini\_Supp(X_j, b, X_k) = \frac{1}{m} \sum_{c \in D_k} r_{jk}(b,c)$$

Approximate supports ($ap\_Supp(X_i, a)$) at an arbitrary node are computed in the following way,

$$ap\_Supp(X_i, a) = \sum_{j \in \mathbf{P}} r_{ij}(a, v^j) + \sum_{j \in \mathbf{F}} ini\_Supp(X_i, a, X_j)$$

When the current subproblem has many pruned values, this approximation can be coarse because it is counting many non-existing contributions. However, at high tree levels, where few assignments have been propagated, one may expect that few values have been pruned. Then, the approximation is supposed to be accurate. Interestengly, it is on the highest tree levels where heuristics have a larger impact in search efficiency because their decisions affect to the largest subproblems. The extreme case occurs at the tree root, where the proposed approximation is equivalent to the exact support.

One efficient way to implement this approximation is to compute gradients at each node by updating the gradient of its parent. This is done with the following procedure,

1. At the tree root, approximate supports are the exact supports

$$ap\_Supp(X_j, b) = Supp(X_j, b) = \sum_{k=1}^{n} ini\_Supp(X_j, b, X_k)$$

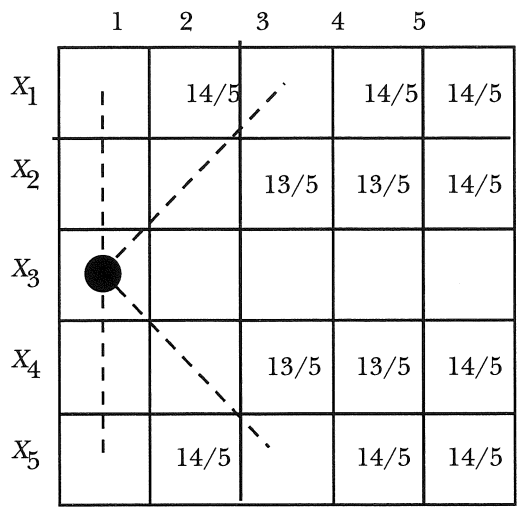
2. At a node S^a that differs from its parent S in that the current variable X_i has been instantiated with value a, we compute the approximate supports $(ap\_Supp^a(X_j, b))$ from the approximate supports at S $(ap\_Supp(X_j, b))$ and the individual contribution of X_i to the initial support $(ini\_Supp(X_j, b, X_i))$ with the following rule

$$ap\_Supp^a(X_j, b) = ap\_Supp(X_j, b) - ini\_Supp(X_j, b, X_i) + r_{ij}(a,b)$$

The *approximate lowest support* variable ordering heuristic (ALS) and the *approximate highest support* value ordering heuristic (AHS) are defined as their exact counterparts, but using approximate supports. With our approach, computing one approximate support is done in constant time. Therefore, the cost of computing all approximate supports is $O(n \cdot m)$, which gives a significant gain in efficiency with respect to exact supports.

Example 6.7:

Consider the 5-queens problem and the approximate heuristics. Initially, approximate supports and exact supports are equivalent. Then, ALS proposes to start assigning X_3 and AHS proposes value 1 as a first choice (like in Example 6.6). Approximate supports at node $\{X_3 \leftarrow 1\}$, are computed by substracting to the initial supports the contribution of X_3 and adding 1. The following table shows the result,



Comparing with exact supports (Example 6.6), one observes that this approximation is not very accurate. The reason is that in this small problem the propagation of a single value causes an important change in domains (all domains are reduced to almost half their size). However, the approximation is supposed to be more accurate on larger problems.

In this situation, ALS selects X_2 and AHS selects to attempt in first place value 5, so the next visited node is $\{X_3 \leftarrow 1, X_2 \leftarrow 5\}$. This decision causes FC to fall in a dead-end. However, it is only needed to visit one more node $\{X_3 \leftarrow 1, X_2 \leftarrow 5, X_4 \leftarrow 4\}$ to detect it. Next, FC backtracks to X_2, attempt value 3 and the approximate heuristics head search straight to solution $\{X_1 \leftarrow 5, X_2 \leftarrow 3, X_3 \leftarrow 1, X_4 \leftarrow 4, X_5 \leftarrow 2\}$, which is the same solution found with exact supports

6.5 Comparison with Other Heuristics[2]

In subsections 2.2.5 and 2.3.4, we introduced different heuristics that appear in the CSP literature. In this section, we show how some of them are closely related to our approach.

For clarity purposes in the comparison, we introduce some new terminology. Consider an arbitrary node such that X_i is a future variable. We denote *past* and *fut* the number of past and future variables; dg_i is the degree of X_i; the number of past variables with which it is connected (backward degree if it becomes the current variable) is denoted bd_i; and

[2]LS, HS and their approximate counterparts were first presented in [Meseguer and Larrosa, 95]. Some approaches discussed in this Section were posterior to that work.

the number of future variables with which it is connected (forward degree if it becomes the current variable) is denoted fd_i. The current domain cardinality of X_i is denoted dom_i . We consider different variable and value ordering heuristics proposed for CSP and MAX-CSP.

6.5.1 Variable Selection in CSP

The most popular variable ordering heuristic for CSP is *minimum domain* (MD) [Haralick and Elliot, 80]. It selects the variable having the minimum number of values in its current domain. MD is unable to discriminate among variables having the same domain cardinality. This fact is especially relevant in problems where all variables have the same number of values because the first choice is made blind. To circunvent this drawback, different approaches have been proposed. For instance, MD-DG [Frost and Dechter, 95] break ties with variable degree and DOM/DG [Bessière and Régin, 96] selects the variable that minimizes the ratio domain cardinality divided by degree. These three heuristics differ among them in the importance that they give to variable degree. MD completely disregards degree, MD-DG gives to degree a secondary role and DOM/DG gives to degree the same importance as to domain size.

The main difference between these heuristics and LS is that they consider all problem constraints as equally important, while LS considers their constraining behaviour. In order to make the comparison, we simplify LS assuming that all variables give the same support s to other variable values with which they are constrained. Then,

$$\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) = s \qquad \forall X_i, X_j \in \mathbf{F}; \forall a \in D_i, \text{ and constrained}(X_i, X_j)$$

$$\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) = 1 \qquad \forall X_i, X_j \in \mathbf{F}; \forall a \in D_i, \text{ and not consr. } (X_i, X_j)$$

With FC, a future value support is,

$$Supp(X_i, a) = past + \sum_{j \in \mathbf{F}} \left(\frac{1}{|Feasible(D_j)|} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) \right)$$

Considering the hypothesis of constant individual supports, this expresion can be rewritten as,

$$Supp(X_i, a) = past + 1 \ (fut - fd_i) + s\, fd_i$$

which can be simplified as,

$$Supp(X_i, a) = n - 1 + fd_i \ (s\text{-}1)$$

LS selects the variable with the lowest sum of supports. For X_i this sum is,

$$\sum_{a \in Feasible(D_i)} Supp(X_i, a) = dom_i\ (n - 1 + fd_i\ (s\text{-}1))$$

Observe that there are two elements contributing to this expresion: the domain cardinality and the variable forward degree. Depending on what value we assume for s, LS has a different behaviour,

(a) If s=1, it means that there is no difference between two variables being constrained or not. In that case the effect of the variable degree becomes null and LS is equivalent to MD.

(b) If s<1 and s is very close to 1, forward degree has a very low negative effect in the expresion. It can only discriminate among variables having the same cardinality. In that case, LS is equivalent to MD-DG with the only exception that LS breaks ties with forward degree instead of full degree.

(c) If s<1 and it is not close to 1, forward degree gains relevance in the expression. In that case, LS is similar to DOM/DG in the sense that combines similar information in a similar way. The differences are that LS uses forward degree, while DOM/DG uses full degree, and that LS uses forward degree as a multiplying negative factor, while DOM/DG uses the inverse of full degree as a multiplying positive factor.

6.5.2 Value Selection in CSP

Different value ordering heuristics have been proposed in the CSP context. Under the influence of planning/scheduling problems, [Keng and Yun, 89] propose to consider values by increasing *cruciality*, where a crucial value corresponds with a highly requested resource. Thus, among different resource alternatives it is preferred the least requested one with the objective of not causing a dead-end. With our notation, an arbitrary value a of the current variable X_i has the following cruciality,

$$Cruciality(X_i, a) = \frac{1}{dom_i} + \sum_{j \in \mathbf{F}} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} (1 - r_{ij}(a,b)) \right)$$

It can be rewritten as,

$$Cruciality(X_i, a) = \frac{1}{dom_i} + fut - \sum_{j \in \mathbf{F}} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) \right)$$

The first element of the summation is common to all values, so it has no effect. Regarding the second element, it is obvious that it is equivalent to select values by increasing cruciality that to select them by decreasing support. Therefore, our HS is equivalent to Keng and Yun cruciality heuristic.

An alternative value ordering heuristic, denoted LVO, was presented in [Frost and Dechter, 95]. They propose to sort values by decreasing number of consistent remaining values. That is,

$$LVO(X_i, a) = \sum_{j \in \mathbf{F}} \sum_{b \in Feasible(D_j)} r_{ij}(a,b)$$

Which follows the same intuition than HS but differs in that the contribution from each variable is the total number of consistent values, rather than the ratio.

Finally, [Geelen, 92] propose to sort values by decreasing *promise*. With our notation, an arbitrary value a of the current variable X_i, it has the following promise,

$$Promise(X_i, a) = \prod_{j \in \mathbf{F}} \sum_{b \in Feasible(D_j)} r_{ij}(a,b)$$

Which again follows the same intuition than HS but has two differences: The contribution from each variable is multiplied instead of added and contributions of individual future variables are absolute numbers instead of ratios.

6.5.3 Variable Selection in MAX-CSP

In [Freuder and Wallace, 92] PFC is combined with a variable ordering heuristic which selects the variable having the largest *inconsistency count mean* (ICM). ICM can be stated in terms of minimizing the following expresion,

$$ICM(X_i) = past - \frac{1}{dom_i} \sum_{a \in Feasible(D_i)} ic_{ia}$$

where ic_{ia} is the current IC computed by PFC. In the MAX-CSP case, supports can be rewritten in the following way,

$$Supp(X_i, a) = past - ic_{ia} + \sum_{j \in \mathbf{F}} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) \right)$$

Therefore, the sum of supports is,

$$\sum_{a \in Feasible(D_i)} Supp(X_i, a) = \sum_{a \in Feasible(D_i)} (past - ic_{ia}) +$$

$$+ \sum_{a \in Feasible(D_i)} \sum_{j \in \mathbf{F}} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b) \right)$$

which can be rewritten as,

$$\sum_{a \in Feasible(D_i)} Supp(X_i, a) = dom_i \left(past - \frac{1}{dom_i} \sum_{a \in Feasible(D_i)} ic_{ia}\right) + \\ + \sum_{a \in Feasible(D_i)} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b)\right)$$

Observe that the second factor of the first term in the summation is the numerical expresion of ICM, so it can be rewritten like,

$$\sum_{a \in Feasible(D_i)} Supp(X_i, a) = dom_i \; ICM(X_i) + \sum_{a \in Feasible(D_i)} \left(\frac{1}{dom_j} \sum_{b \in Feasible(D_j)} r_{ij}(a,b)\right)$$

Therefore, we see that LS is a refinement of ICM considering two additional sources of information: from the first element of the summation, we see that LS considers the domain cardinality and prefers variables having few feasible values; from the second element of the summation, we see that LS prefers variables whose values are highly constrained with future values.

6.5.4 Value Selection in MAX-CSP

There are three value ordering heuristics for MAX-CSP often used in the literature. In [Freuder and Wallace, 92] values are selected by increasing number of inconsistencies with past variables (we call this heuristic IC). In [Wallace and Freuder, 93] values are selected by increasing number of arc-inconsistencies which are computed during a pre-processing step (we call this heuristic ACC). In [Larrosa and Meseguer, 96] values are selected by increasing number of inconsistencies with past variables plus arc-inconsistencies with future variables (we call this heuristic IC+DAC). These three heuristics can be seen as simplifications of HS. For instance, IC is equivalent to HS if the contribution of future variables to supports are discarded. IC+DAC is equivalent to HS if the contribution of non arc-inconsistent future variables to supports are discarded. Finally, ACC is equivalent to HS if the contribution of non arc-inconsistent variables to supports are discarded, no matter whether they are future or past variables.

6.6 Experimental Results

6.6.1 Total Constraint Satisfaction

The first set of experiments endeavours to evaluate the effectiveness of our heuristics in total constraint satisfaction. In these experiments we used the following classes of random problems,

(a). <35, 6, 501/595, p_2>	(b). <35, 9, 178/595, p_2>
(c). <50, 6, 325/1225, p_2>	(d). <50, 20, 95/1225, p_2>
(e). <125, 3, p_1, 1/9>	(f). <350, 3, p_1, 1/9>

All these problem classes have been already used for evaluation purposes in the literature [Bessière and Régin, 96]. For each parameter setting we generated samples of 50 instances.

We solved each problem using FC with four different heuristic combinations:

1. The first combination is *minimum domains* as variable selection and lexicographical ordering as value ordering. We will refer to this combination as MD_LEX. This is a combination widely used with random problems and we use it as a reference.
2. The second combination is *domain size divided by degree* for variable selection [Bessière and Régin, 96] and *look-ahead value ordering* for value ordering [Frost and Dechter, 95]. We will refer to this combination as DOM/DG_LVO. It is believed that this combination is one of the most effective for random problems [Bessière and Régin, 96].
3. The third combination is *approximate lowest support* variable selection and *exact highest support* value ordering. We will refer to this combination as ALS_HS. This is a fair competitor with DOM/DG_LVO because it performs a similar amount of computation at each node. In addition, we saw in the previous Section that these heuristics are closely related. Thus, a similar performance can be expected.
4. The fourth combination is *approximate lowest support* variable selection and *approximate highest support* value ordering. We will refer to this combination as ALS_AHS. The purpose of this combination is to quantify the cost associated to the accuracy decrement of using approximate supports in value ordering.

We have not tried the combination LS_HS because the cost of LS is prohibitive for random problems.

Figures 6.3-6.8 report the results of our experiments. Each figure corresponds to a problem class and reports the average search cost in terms

of consistency checks, visited nodes and CPU time. For problem classes where tightness is the varying parameter (*i.e.*: from (*a*) to (*d*)), we only plot the region that corresponds to the complexity peak. Figure 6.3 presents the results with the <35, 6, 501/595, p_2> class. Regarding the number of visited nodes, we see that DOM/DG_LVO and support-based heuristics give a gain ration of about 1.5 over MD_LEX. Support-based heuristics are slightly better than DOM/DG_LVO. Interestingly, approximating supports for value ordering does not cause any performance loss. Regarding the number of checks, we observe that the cost of LVO and HS does not pay off its tree reduction. Nevertheless, the consistency-check-free nature of AHS makes this choice the most cost effective. Regarding CPU time, we see that DOM/DG_LVO and ALS_HS are not cost effective and ALS_AHS gives a reduced (althouth worthwhile) gain.

Figures 6.4 and 6.5 present a similar behaviour in the <35, 9, 178/595, p_2> and <50, 6, 325/1225, p_2> problem classes. Regarding number of nodes, we again see that DOM/DG_LVO and support-based heuristics clearly outperform MD_LEX. The tree reduction goes up to a factor of 4. Support-based heuristics and DOM/DG_LVO present a similar performance. Like in the previous case, approximating supports for value ordering does not cause any performance loss. Regarding the number of checks, we observe that support-based heuristics and DOM/DG_LVO are considerably more efficient than MD_LEX. However, only ALS_AHS achieves a gain ratio of nearly 4, which is equal to its gain ratio in terms of visited nodes. Regarding CPU time, support-based heuristics and DOM/DG_LVO are nearly twice faster than MD_LEX. In random problems, performing a consistency check is cheap. For this reason, the ratio gain that ALS_AHS has over ALS_HS and DOM/DG_LVO in terms of checks is not maintained in terms of CPU time. Nevertheless, AHS is supposed to be more fruitful in domains where performing consistency checks is more costly.

Figure 6.6 reports our results on the <50, 20, 95/1225, p_2> class. Independently of what search effort measurement is considered, DOM/DG_LVO is much more effective than the other heuristic combinations. Nevertheless, our support-based heuristics clearly outperform MD_LEX. Regarding CPU time, the gain ratio of DOM/DG_LVO over MD_LEX goes up to 50 and the gain ratio of DOM/DG_LVO over ALS_HS goes up to 6. In this case, AHS outperforms HS in terms of checks, but HS outperforms AHS in terms of nodes and time. Again, this fact can be understood because consistency checks in random problems are cheap.

Figures 6.7 and 6.8 present the results of the <125, 3, p_1, 1/9> and <350, 3, p_1, 1/9> problem classes. For this problems, the relative performance between DOM/DG_LVO and support-based heuristics is reversed with respect Figure 6.6. The combination presenting the worst performance is MD_LEX. Independently of what search effort measurement is considered, ALS_AHS is the best choice (except in terms of visited nodes, where ALS_AHS and ALS_HS have practically the same

performance). Comparing with DOM/DG_LVO in the <125, 3, p_1, 1/9> class, ALS_AHS is about 5 times better in terms of checks, 2 times better in terms of visited nodes and 1.5 times better in terms of CPU time. Comparing with DOM/DG_LVO in the <350, 3, p_1, 1/9> class, ALS_AHS is about 25 times better in terms of checks and visited nodes and 15 times better in terms of CPU time.

From this experiments, we see that both support-based heuristics and DOM/DG_LVO are, in general, much better than MD_LEX (although there are exceptions like the <35, 6, 501/595, p_2> class where MD_LEX is slightly better in terms of CPU time). Comparing DOM/DG_LVO, there is no clear winner. In some problem classes, both combinations present a quite similar performance. This result could be expected from the analysis performed in Section 6.5, where we show that DOM/DG is similar than a simplified version of LS, and LVO is similar than HS. However, one can find problem classes where one combination is clearly better than the other and the way around. Regarding HS vs. AHS, we observed that the approximation can save many consistency checks without a heuristic quality degradation (the number of visited nodes is, in general, similar). It has to be mentioned, that the same idea used to approximate HS can be applied to LVO and one can expect a similar behaviour.

6.6.2 Partial Constraint Satisfaction

The second set of experiments endeavours to evaluate the effectiveness of our heuristics in partial constraint satisfaction. In this context, there is not a consensuated variable ordering heuristic that can be used as a reference (such as MD in total constraint satisfaction). Therefore, we performed an initial experiment with the <15, 10, 50/105, p_2> problem class. For each parameter setting we generated samples of 10 instances. We solved each problem using PFC-RDAC (described in Section 4.7) with five different variable ordering heuristic (values were always sorted by IC+DAC):

1. FD/BD: *forward degree/ backward degree.* This is a static variable ordering which selects first the variable that has most constraints with already chosen variables. It breaks ties preferring variables that have most constraints with still not chosen variable. This heuristic was found in [Larrosa and Meseguer, 96] the most effective static variable ordering among several alternatives.
2. ICM: it selects the variable having the largest *inconsistency count mean* [Freuder and Wallace, 92].
3. MD: like in total constraint satisfaction, it selects the variable having the minimum domain size.
4. MD-DG: as in total constraint satisfaction, it selects the variable having the minimum domain size using variable degree as a tie breaker. This heuristic was found the most effective heuristic for

PFC-RDAC among several alternatives for the experiments presented in Chapter 4.

5. ALS: *approximate lowest support* as described in Section 6.4.

Figure 6.9 shows the results of this experiment in terms of visited nodes (checks and time give similar results). It can be observed that LM and MD are by far outperformed by the rest of heuristics. We also see that the recognized dominance of dynamic over static orderings in total constraint satisfaction cannot be extended to MAX-CSP (we already observed this fact in Chapter 4). Nevertheless, dynamic orderings seem to be slightly better than FD/BD. The two heuristics that give the best performance are MD-DG and ALS.

For the remaining experiments, we select the best variable orderings found above and make a more exhaustive comparison combining them with value ordering heuristics. We used the same classes of random problems that we already used in Chapter 4,

(a). <10, 10, 45/45, p_2>	(b). <15, 5, 105/105, p_2>
(c). <15, 10, 50/105, p_2>	(d). <20, 5, 100/190, p_2>
(e). <25, 10, 37/300, p_2>	(f). <40, 5, 55/780, p_2>

Recall that (a) and (b) are highly connected, (c) and (d) are problems with medium connectivity, and (e) and (f) are sparse problems. For each parameter setting, we generated samples of 50 instances.

Each problem was solved with PFC-RDAC and two different combinations of heuristics: The first combination is MD-DG as variable selection and IC+DAC as value ordering and the second combination is ALS with AHS.

Figures 6.10-6.15 report the results of these experiments. Each figure corresponds to a problem class and includes three plots reporting the average search cost in terms of consistency checks, visited nodes and CPU time. Since both combinations perform a similar amount of work at each node, the three measurements are very correlated. It can be seen that both heuristic combinations give a very close performance, although ALS_AHS is slightly better (except for the tightest dense problems). ALS_AHS seems to produce larger gains in sparse problems and in problems with low tightness. The highest gain ratios occur in sparse problems where ALS_AHS is typically twice faster than MD-DG_IC+DAC.

6.7 Conclusions and Future Work

From the work presented in this Chapter, we extract several conclusions. Although LP and CSP seem similar, we have shown that there is a major difference between them. Constraint satisfaction is a global optimization

task, while LP is a local optimization task. Our analysis has served for a better understanding of the relationship between these two formalisms.

The effect that variable and value orderings have in algorithsms for CSP and MAX-CSP is not equivalent. However, heuristics for both cases can be thought in the same terms. We have seen that the fail-first and succeed-first principles provide the basic targets for variable and value ordering heuristics. Interestingly, this principles are valid for both CSP and MAX-CSP.

Depth-first search and local optimization are completely different algorithmic schemas. However, a local optimization perspective can be useful to heuristically guide depth-first search. Most previous heuristics are obtained by pure experimental means. Our heuristics, which are based on local gradients, can be reduced to them under certain assumptions. In that sense, our approach gives insight into their nature.

Our work raises several questions that deserve further research. The global optimization perspective of constraint satisfaction suggests a closer look to the $A(V)$ function. For instance, it is interesting to study under which conditions $A(V)$ is a convex function because it would give condition for CSP and MAX-CSP tractability.

Stochastic search is an alternative to depth-first which has shown to be very effective in some situations. A well-known disadvantage of stochastic search is that it takes decisions based on *very* local information (the current assignment). Support-based heuristics may be suitable in this context if we give non-zero weight to unassigned values. With this idea supports would take into account the compatibility of values which are not currently assigned.

With our approach, labellings have a homogeneous distribution of weights among future values. It means that we consider every value equally important. However, we may give a higher weight to those values that we believe that are more important (may be based on previously gathered knowledge about the problem).

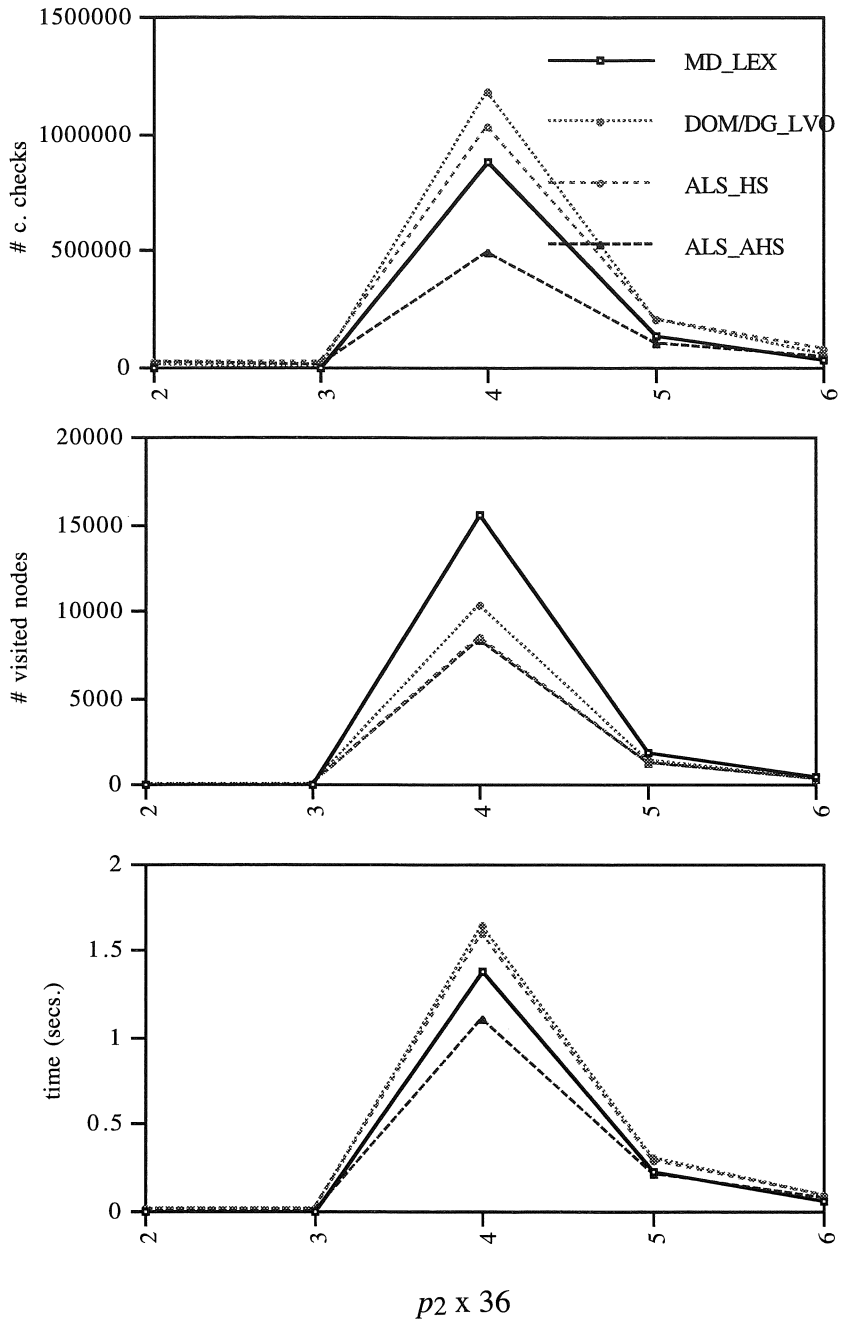


Figure 6.3: Experimental results on the class <35, 6, 501/595, p_2> with different heuristics.

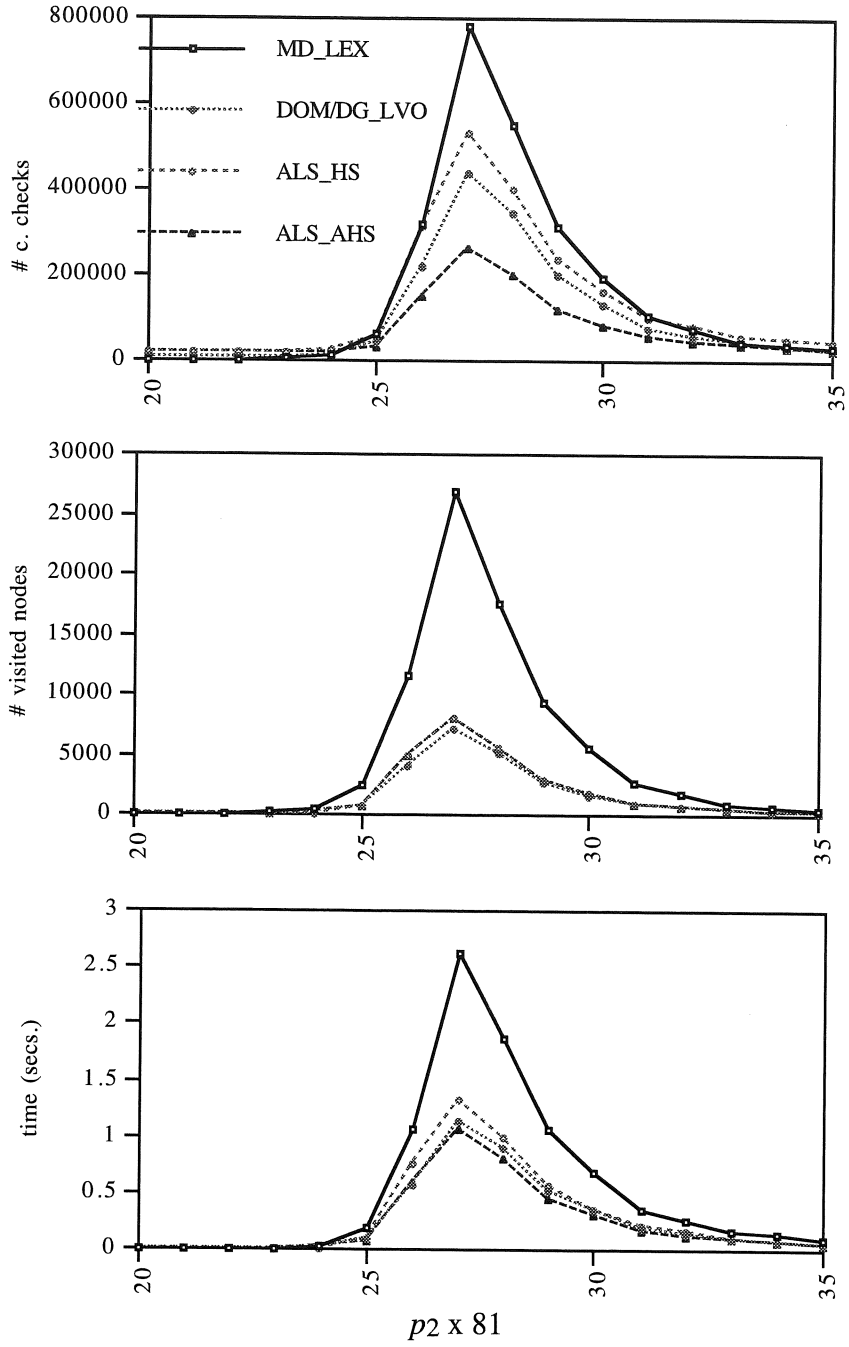


Figure 6.4: Experimental results on the class <35, 9, 178/595, p_2> with different heuristics.

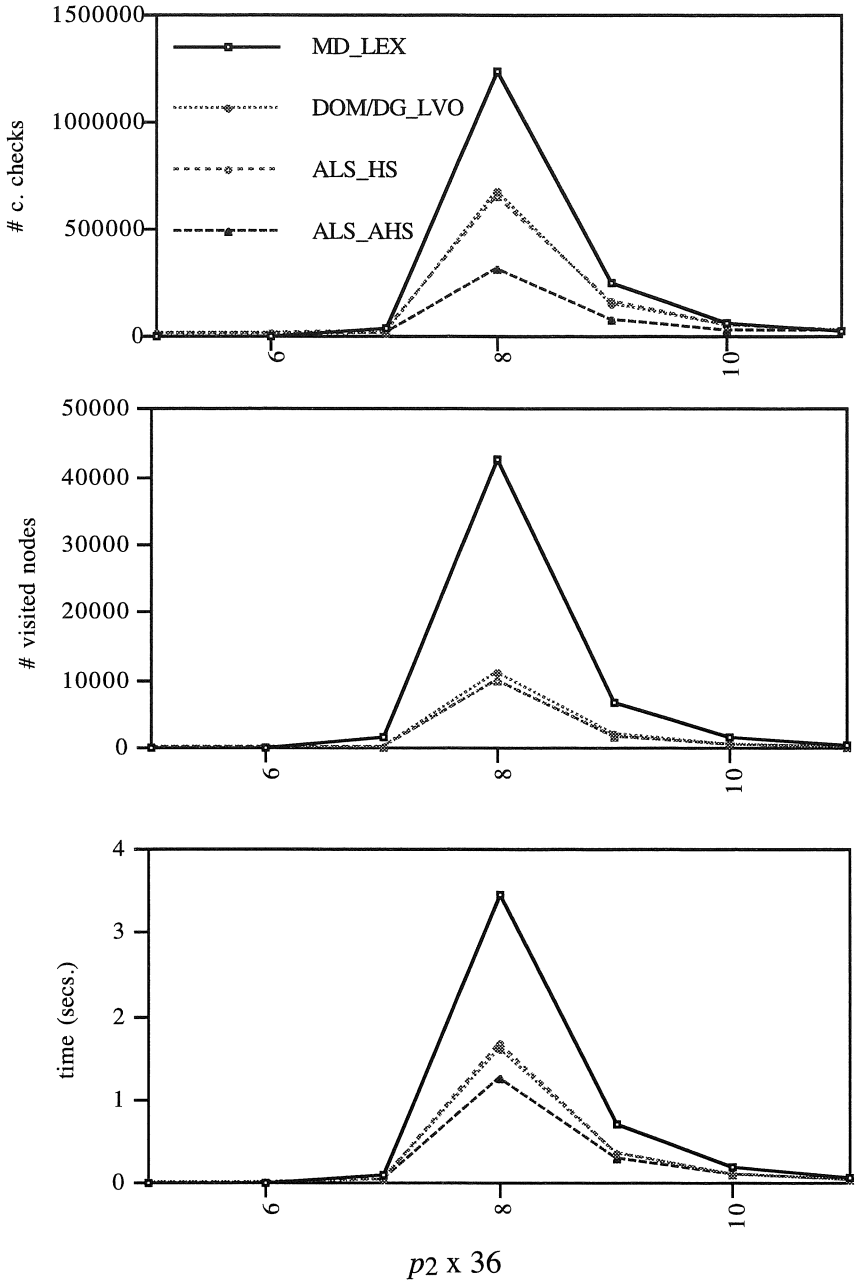


Figure 6.5. Experimental results on the class <50, 6, 325/1225, p_2> with different heuristics.

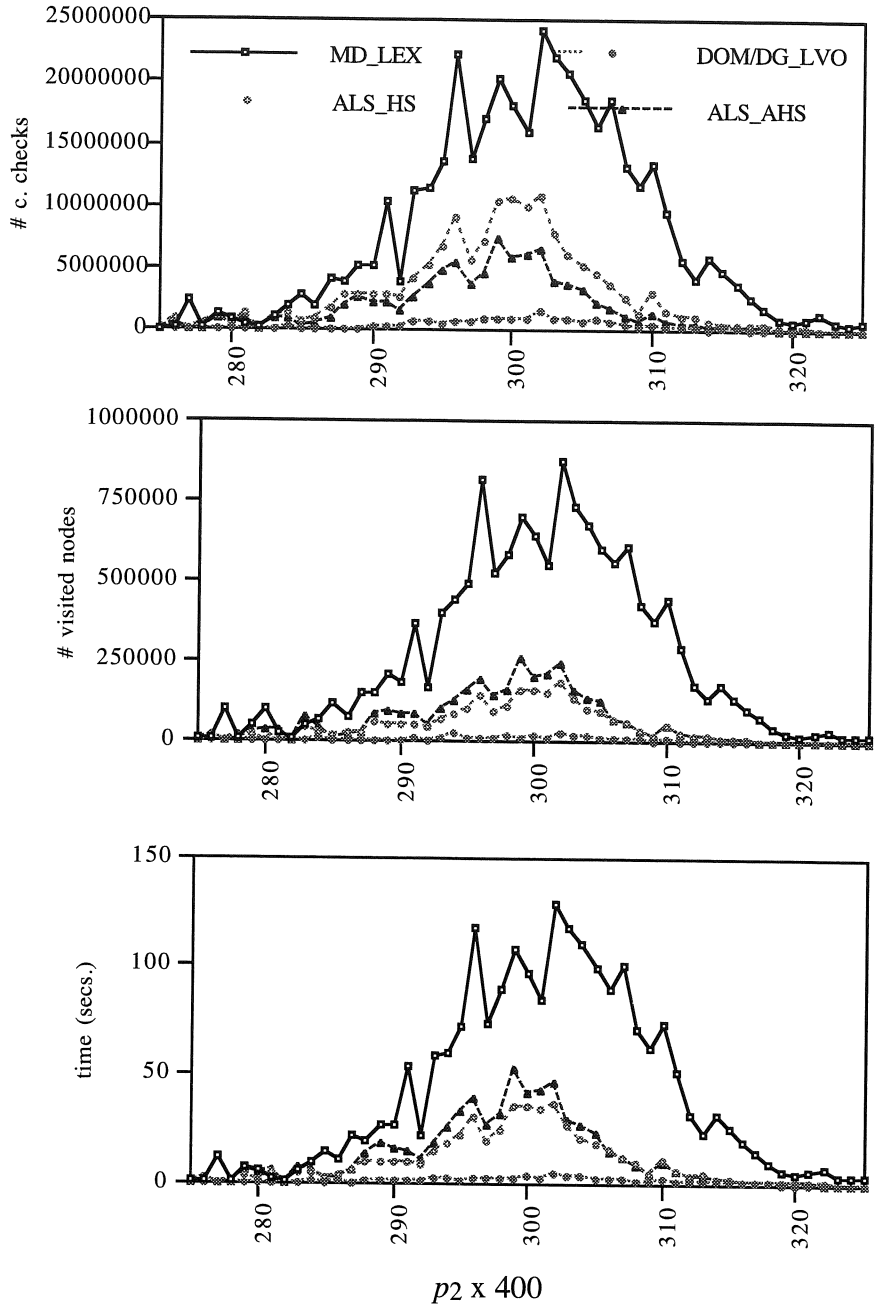


Figure 6.6: Experimental results on the class <50, 20, 95/1225, p_2> with different heuristics.

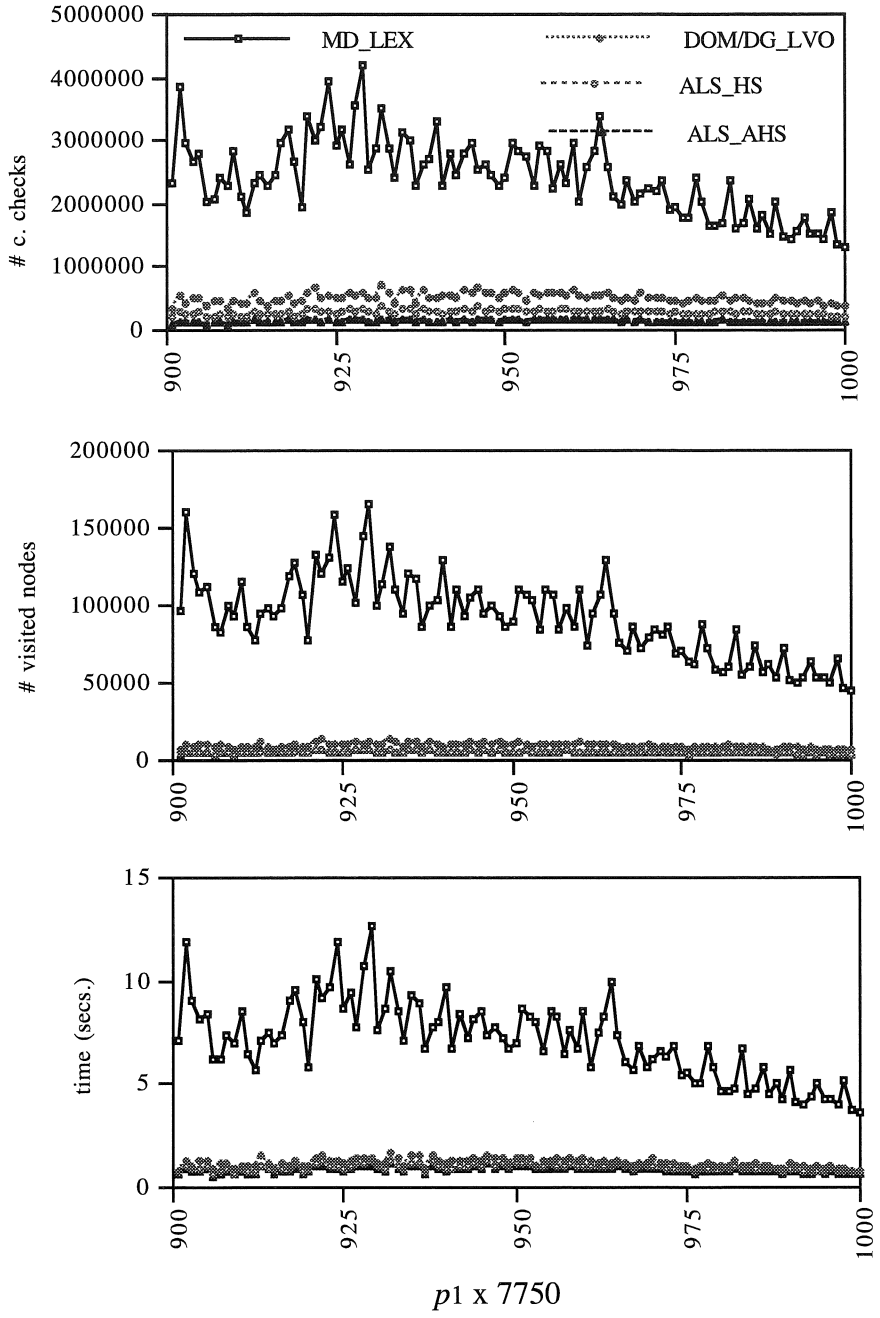
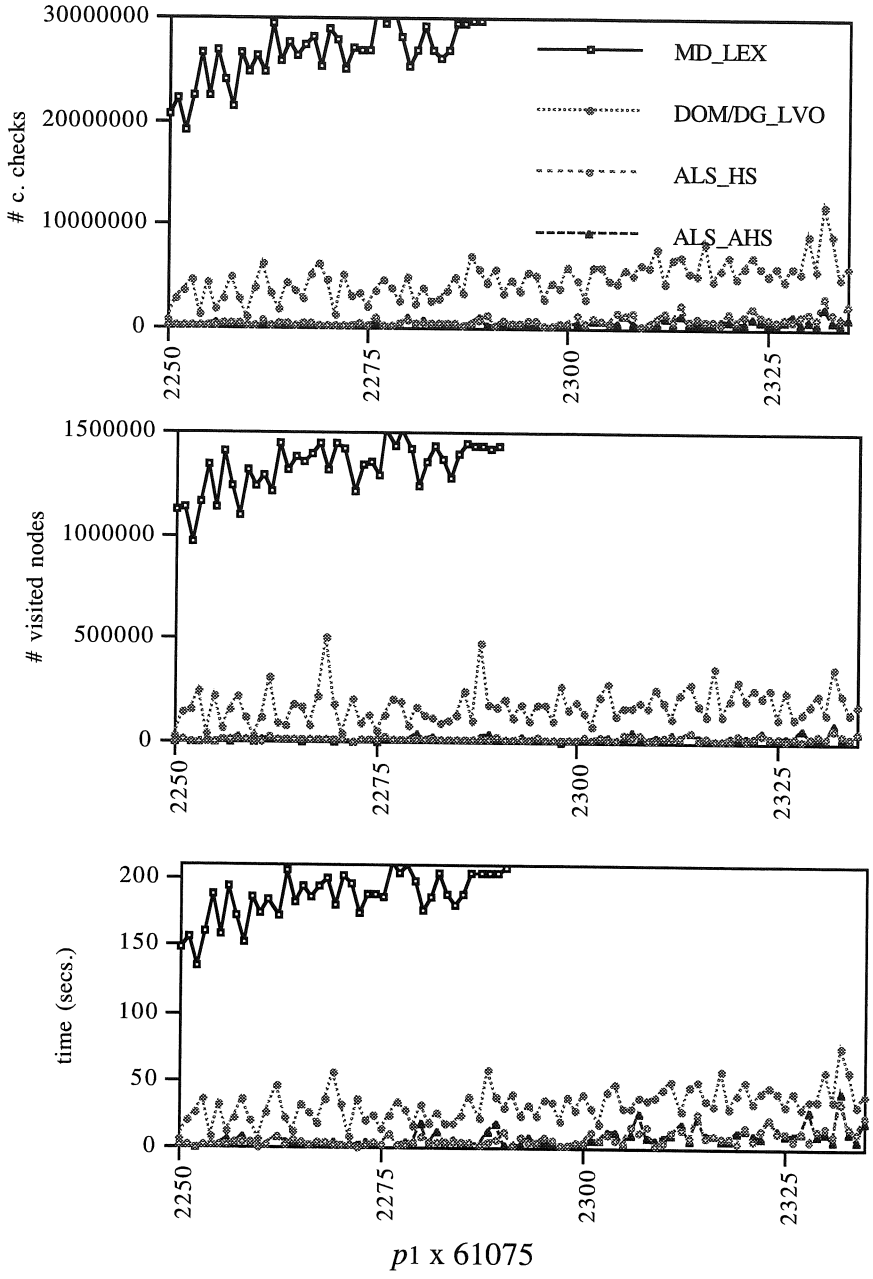


Figure 6.7: Experimental results on the class <125, 3, p_1, 1/9> with different heuristics.


30000000
20000000
10000000
0
c. checks
MD_LEX
DOM/DG_LVO
ALS_HS
ALS_AHS
2250
2275
2300
2325
1500000
1000000
500000
0
visited nodes
2250
2275
2300
2325
200
150
100
50
0
time (secs.)
2250
2275
2300
2325
p1 x 61075

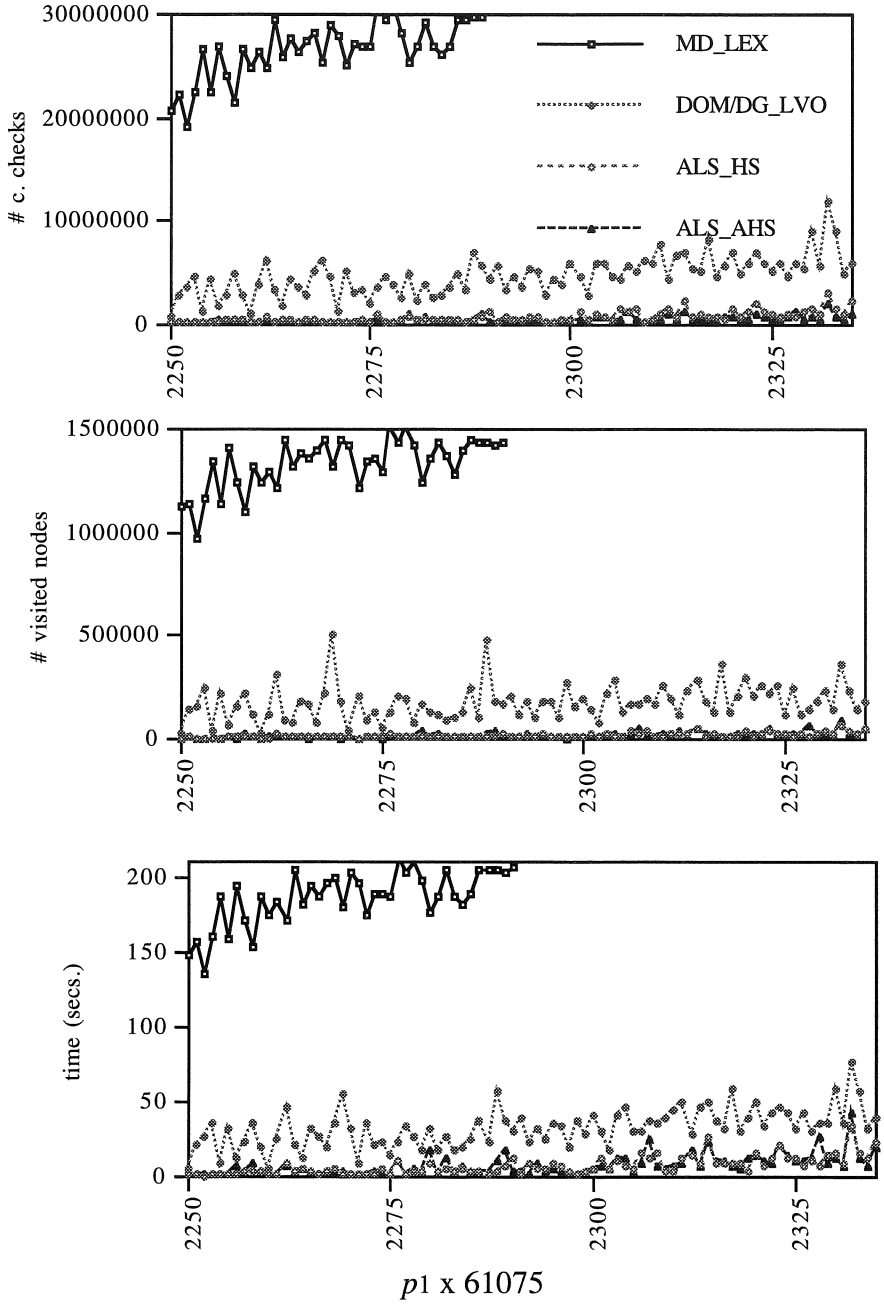


Figure 6.8: Experimental results on the class <350, 3, p_1, 1/9> with different heuristics.

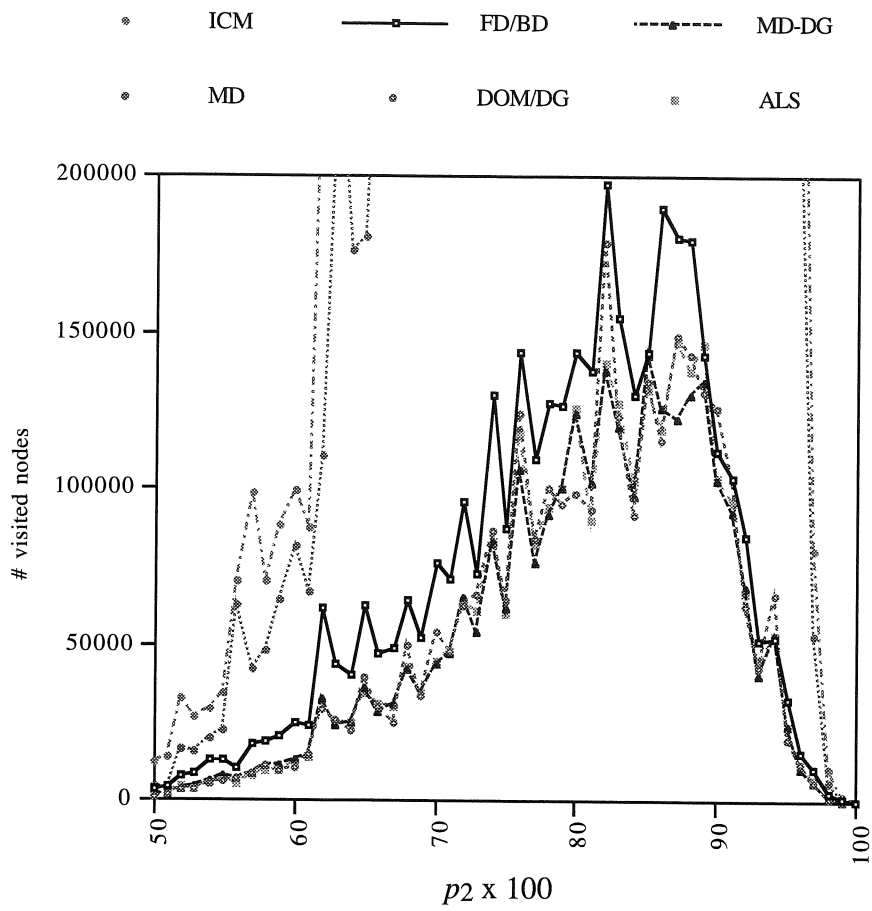


Figure 6.9: Number of visited nodes on the class <15,10,50/105,p_2> with different variable ordering heuristics.

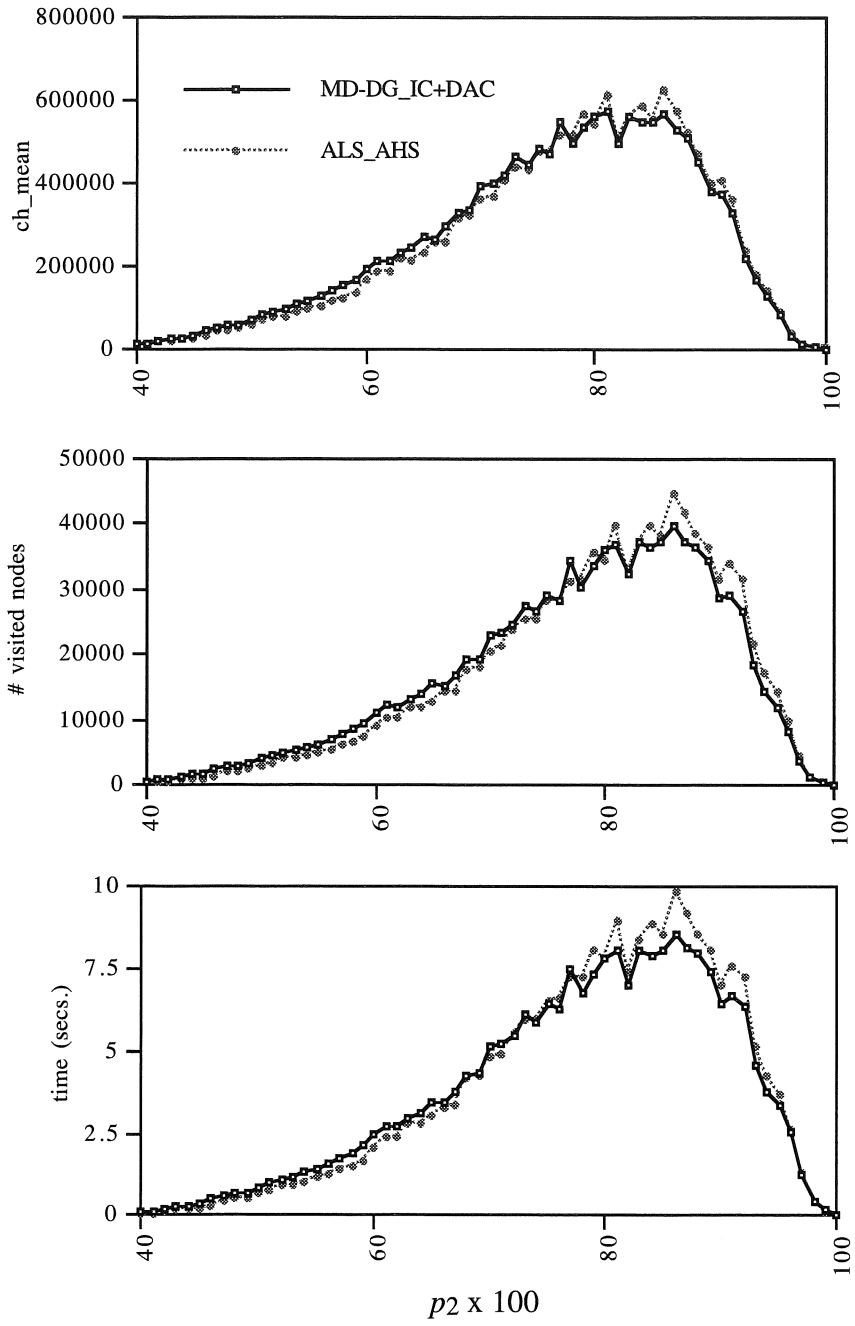


Figure 6.10: Experimental results on the class <10, 10, 45/45, p_2> with two different heuristic combinations.

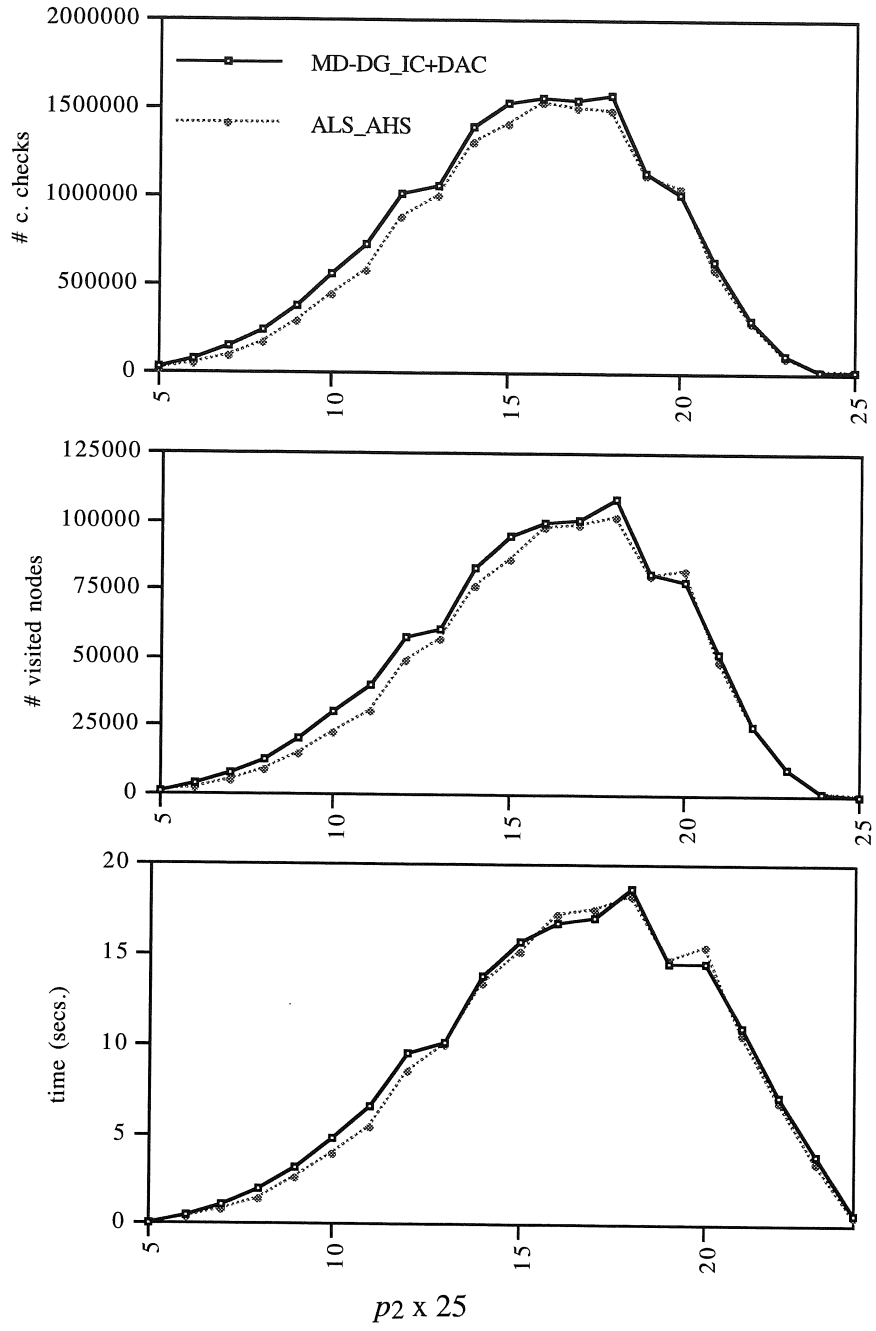


Figure 6.11: Experimental results on the class <15, 5, 105/105, p_2> with two different heuristic combinations.

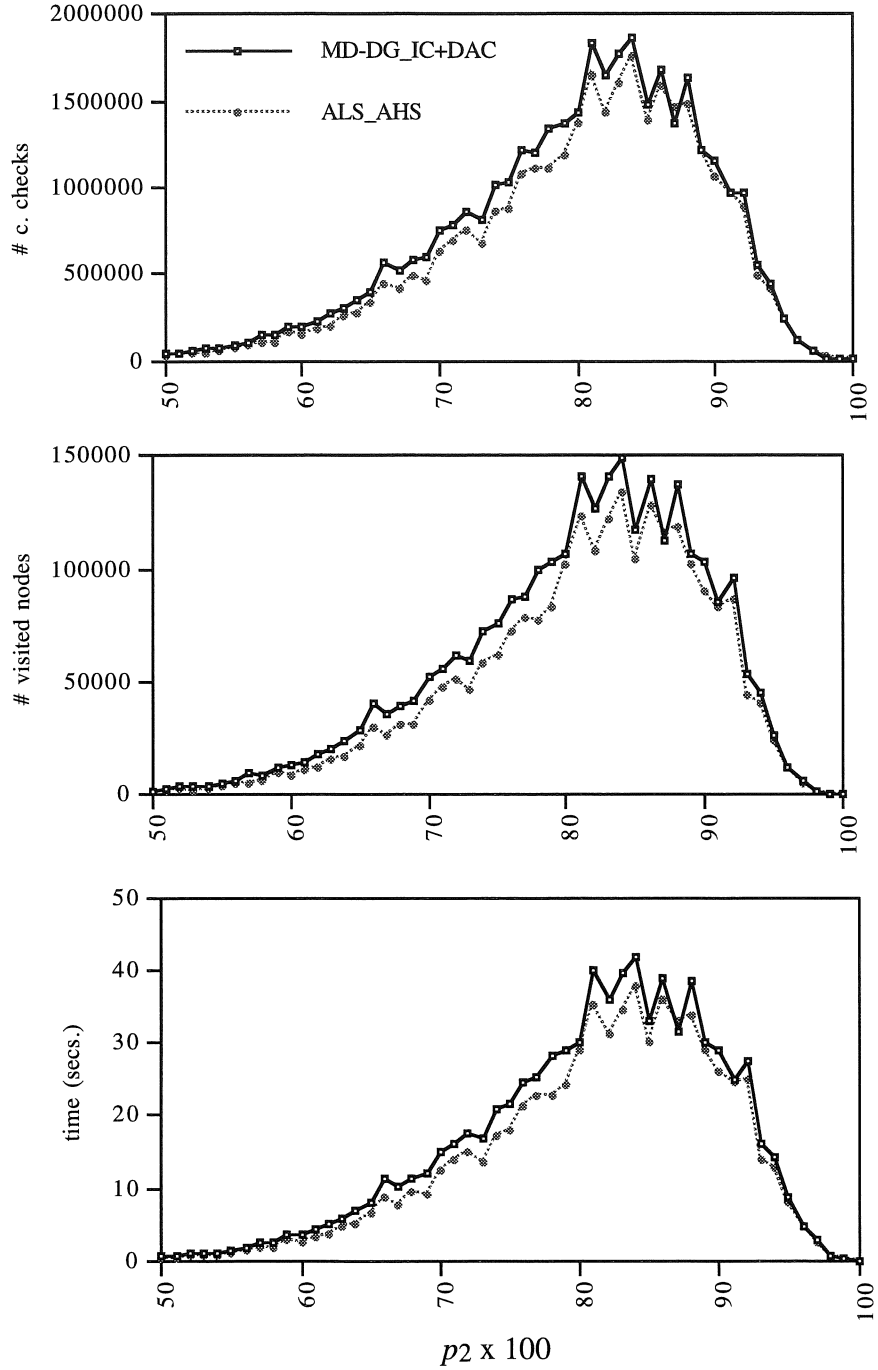


Figure 6.12: Experimental results on the class <15, 10, 50/105, p_2> with two different heuristic combinations.

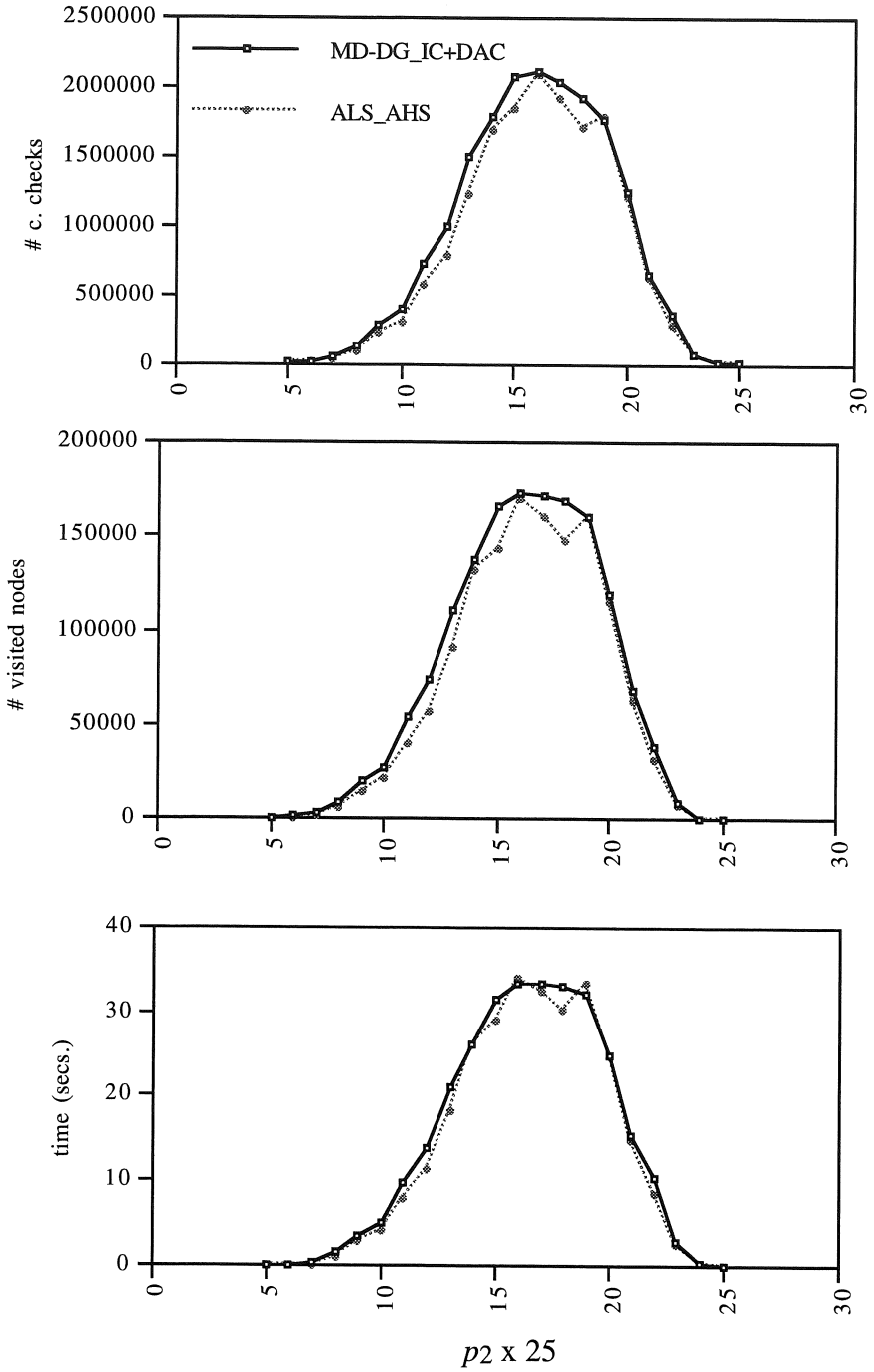


Figure 6.13: Experimental results on the class <20, 5, 100/190, p_2> with two different heuristic combinations.

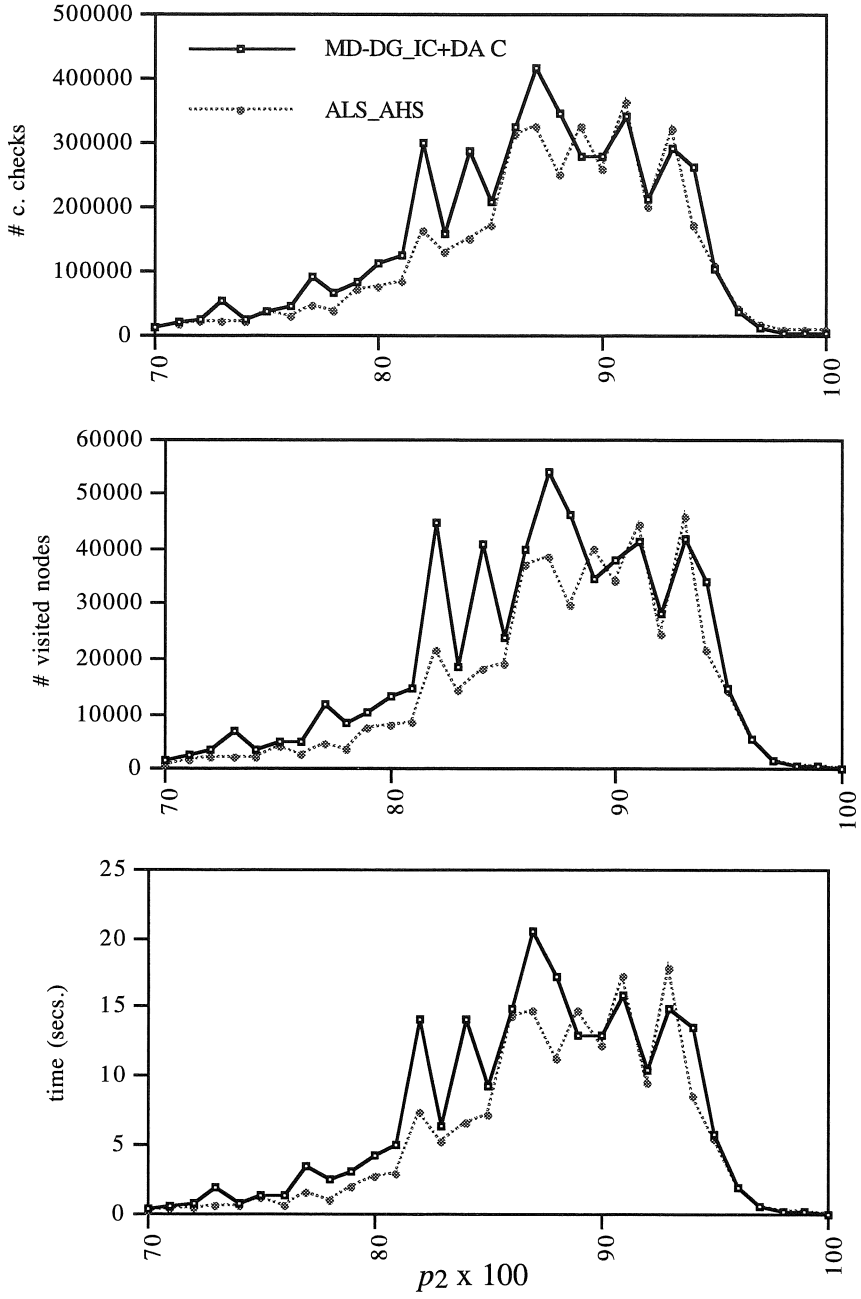


Figure 6.14: Experimental results on the class <25, 10, 37/300, p_2> with two different heuristic combinations.

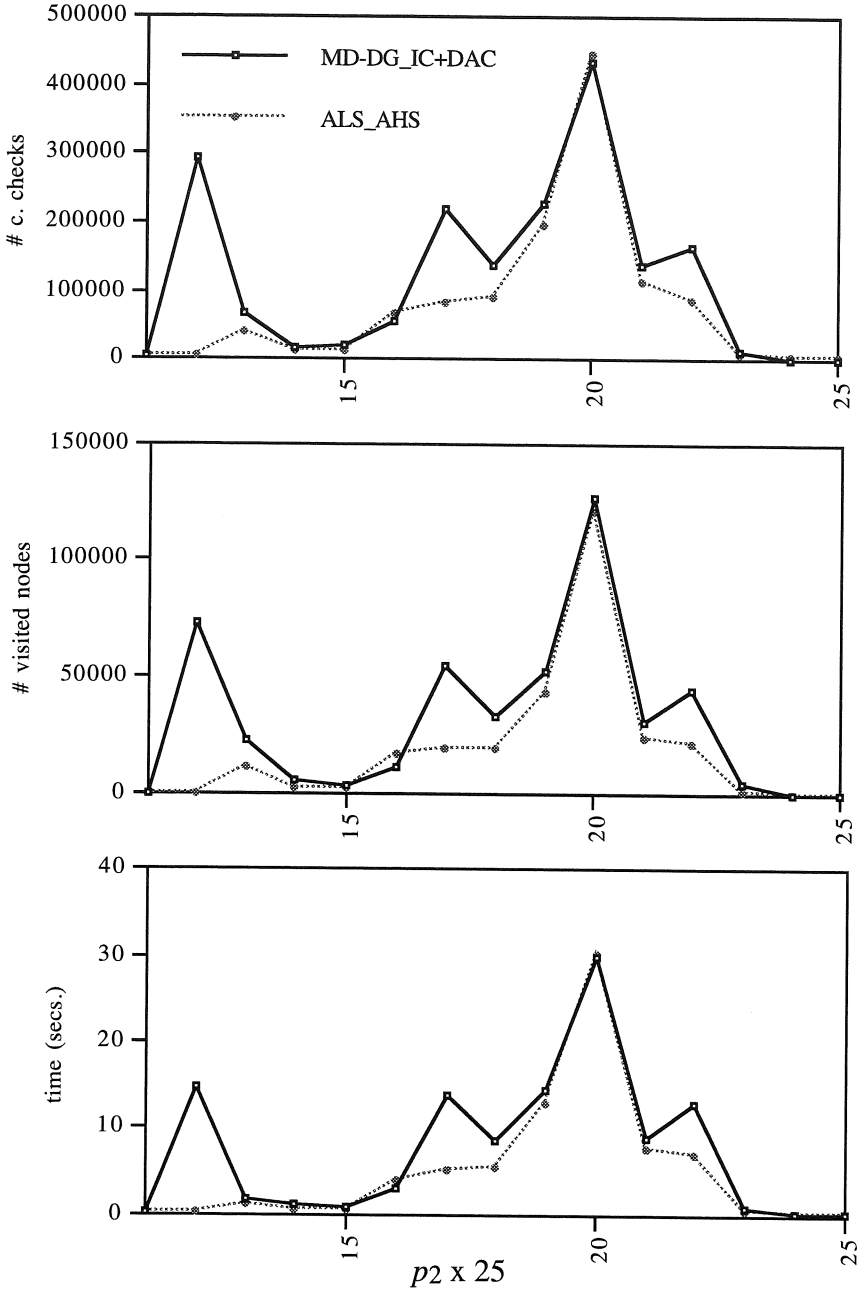


Figure 6.15: Experimental results on the class <40, 5, 55/780, p_2> with two different heuristic combinations.

Chapter 7

Experimental Results on the Job-shop Problem

In this Chapter, we evaluate support-based heuristics —developed in the previous Chapter— in a real application domain. We experimentally show that our heuristics are suitable for the job-shop problem, a classical scheduling problem for which a considerable research effort has been devoted. More precisely, we show that our heuristics are competitive with respect to specific techniques for the job-shop. The interest of this result is that our approach was developed under a general-purpose motivation and does not include any domain-dependent element or any parameter that has to be adjusted manually for this particular domain. In that sense, our generic approach is more robust and more applicable to other problem instances than specific approaches.

This paper is organized as follows. In Section 7.1, we introduce the job-shop problem. In Section 7.2, we revise the last approaches to the job-shop from AI. In Section 7.3, we express the job-shop as a CSP and present two different solving approaches. In section 7.4, we show how support-based heuristics are used with the different approaches. In Section 7.5, we solve both approaches using depth-first based algorithms. In Section 7.6, we solve again both formulations using discrepancy-based algorithms, a new type of search algorithms recently developed. Finally, in Section 7.7, we present the conclusions of this Chapter.

7.1 Introduction

The *job-shop problem* involves the temporal synchronization of the production of a set of jobs. Each job is composed by a sequence of operations; each operation has a duration and it requires the exclusive use of a machine for its duration. Each job has a release date and a due date between which it should be accomplished. This problem can be formulated

as an optimization problem or as a decision problem. In this Chapter, we consider the decision problem, often called the *job-shop with non-relaxable time windows*. A solution is a temporal assignment of operations to machines in such a way that jobs are perfomed in time, satisfying the sequence of its operations and respecting that any machine is used by at most one operation at any time.

The job-shop is a classical problem in manufacturing to which a considerable amount of research has been devoted in different fields of Computer Science. The interest for this problem is not purely academic, since it represents many real-world problems that arise daily in factories and workshops. Simpler, easier and more efficient solving methods are of great interest because of the practical implications that they have. From an Artificial Intelligence perspective, the job-shop has been treated as a CSP by several authors who have developed efficient solving approaches. However, these approaches are specific for this particular domain. In this context, one may believe that generic search and CSP methods are not appropriated for this problem. In this Chapter we contradict this belief. We show that support-based heuristics combined with general purpose search schemas can be successfully applied to the job-shop.

Our claim is based on experimental results on a classical job-shop benchmark [Sadeh, 91; Sadeh and Fox, 96]. Currently, all its problem instances have been solved. However, this benchmark has been considered as challenging in the job-shop community and has attrackted the interest of several researchers. Therefore, it provides an interesting test-case for comparing our work with different approaches. In our experiments, we obtain similar results to specific approaches in terms of the number of solved problems. Our approach does not outperform specific approaches in terms of CPU time, but it reaches a reasonable performance using generic search and CSP methods which, in general, are easier to develop, codify and maintain than specialized approaches.

7.2 Previous Work

The job-shop problem has been object of intense research from different perspectives. In the following, we summarize some recent approaches from an AI point of view.

An important contribution is due to Sadeh and colleagues [Sadeh et al, 95; Sadeh and Fox, 96]. They formulate the job-shop as a CSP, where variables are associated with operations, variable domains are possible start times, and constraints involve precedence among operations and exclusivity in resource use. To solve this problem, they achieve initial arc-consistency on precedence constraints, and they use forward checking plus a weak form of arc-consistency on resource constraints as the basic algorithm. Claiming that "generic CSP variable and value ordering

heuristics do not perform well in the job-shop domain"[1], they develop two specific variable and value ordering heuristics based on resource contention, denominated ORR and FSS respectively. With this approach, they solve 52 out of 60 problems of their benchmark outperforming existing approaches at year 1991. Next year, they were able to solve all the problems of the benchmark. They modified the algorithm enhancing it with three new features:

1. *Dynamic consistency enforcement,* which dynamically identifies critical subproblems and determines how far to backtrack by selectively enforcing higher levels of consistency among variables participating in these critical subproblems.
2. *Learning ordering from failure,* which dynamically modifies the order in which variables are instantiated based on earlier conflicts.
3. *Incomplete backjumping heuristic,* which abandones areas of the search space that appear to require excessive computational effort. This feature renders the algorithm incomplete.

Coincidentally, another approach [Muscettola, 94] was able to solve the 60 problems using a stochastic search procedure with a global heuristic based on resource contention. Resource contention is approximated via Monte-Carlo simulation. If the procedure fails to find a feasible solution, it restarts from scratch, relying on the stochasticity of its Monte-Carlo simulation to produce a different solution.

Alternatively, Smith and Cheng [Smith and Cheng, 93] formulate the problem as a search in a binary decision tree, where each node represents two operations that compete for the same resource. A node has two successors, the two possible orderings for two operations. When one of the orderings is selected, this decision is propagated over the possible start and finish times of all other operations. In addition to this problem formulation, the main contribution of this approach is a dynamic variable ordering heuristic (which pair of operations consider next) and a dynamic value ordering heuristic (which operation post first), based on slacks (free period) left by two operations competing for the same resource, when they are scheduled from their earliest start time and consecutively. These heuristics are used in a very simple search method (called PCP), which selects two operations, determines which one is scheduled first and propagates the effect of this decision. If no operation appears unfeasible, the process iterates, otherwise it stops and returns false (no backtracking is done). With this approach they solve 56 out of 60 Sadeh problems. Using a more complex version of their heuristics, modified in an ad-hoc manner, they are able to solve the 60 problems.

Finally, Crawford and Baker [Crawford and Baker, 94] codify Sadeh's benchmark as propositional satisfiability problems (SAT), and they solve them using a complete algorithm (Davis-Putnam procedure) and two

[1] This was probably true when this research was done, at year 1991. Nowadays it is no longer true, as we will see in this Chapter.

incomplete ones (GSAT and ISAMP). Only ISAMP is able to solve the 60 problems, with a upper limit of 20,000 tries.

From this brief analysis, we can identify complex specialized algorithms such as a modified forward checking with incomplete backtracking [Sadeh et al, 95], complex heuristics based on global resource contention (ORR/FSS [Sadeh and Fox, 96], CPS [Muscettola, 94]), or specialized local heuristics (slack-based [Smith & Cheng, 93]). Only the formulation of Crawford and Baker seems to be generic. However, this approach also presents some drawbacks, and it is the size of the SAT translation of a problem. According to [Harvey, 95], a typical benchmark problem is translated into a theory of 100,000 clauses and 20,000 literals, using more than 1 Mbyte of memory, which seems not to be very operational from a practical perspective. Regarding completeness, all approaches solving the whole benchmark are incomplete (although PCP can be completed easily). For all this, it seems to be a plausible goal to look for simple, generic solving methods for the job-shop, easy to implement and debug, which could solve the problem with reasonable performance.

7.3 The Job-shop as a CSP

The job-shop requires the scheduling of a set of jobs on a set of physical resources. Each job consist of a set of operations to be scheduled according to a process routing that specifies a partial ordering among these operations. In the job-shop considered in this work, process routings are sequences of operations. Each operation (O_i) has an *earliest start time* (est_i), a *latest finish time* (lft_i) and a *duration* (d_i). In addition, each operation requires the exclussive use of a unique *machine* (r_i) during its execution. The objective is to come up with a *start time* (st_i) for each operation such that problem constraints are fulfilled. Figure 7.1 shows a simple job-shop problem which is formed by two jobs, each one including three operations. Each node corresponds to an operation. Simple arrows indicate precedences impossed by the process routing and double arrows indicate different operations competing for the same machine. In addition, at each node we include the earliest-start-time/latest-finish-time interval and the duration of the corresponding operation.

Representing the job-shop as a CSP, variables are operation start times (st_i), domains (D_i) are determined by each operation earliest start time (est_i), latest finish time (lft_i) and duration (d_i), D_i=[est_i, lft_i-d_i]. There are two different kinds of constraints:

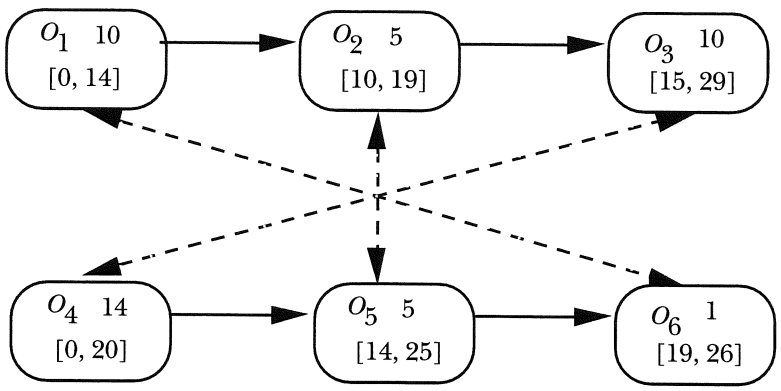
(a). *Precedence* constraints (between consecutive operations of a job): if operation i must be executed before operation j, then $st_i + d_i \leq st_j$.

(b). *Resource* constraints: if operations i and j require the same machine, then $st_i + d_i \leq st_j$ or $st_j + d_j \leq st_i$.

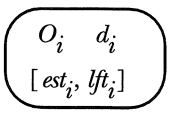
For instance, in the example of Figure 7.1 operations O_1 and O_2 are constrained by a precedence constraint $R_{12}=\{(t,t')|\ t + 10 \leq t'\}$ and the assignment $\{st_1 \leftarrow 1,\ st_2 \leftarrow 15\}$ is consistent with respect that constraint (t and t' are potential start times for the two considered operations). Operations O_2 and O_5 compete for the same machine. Therefore, they are constrained by a precedence constraint $R_{25}=\{(t,t')|\ t + 5 \leq t' \text{ or } t' + 5 \leq t\}$ and the assignment $\{st_2 \leftarrow 10,\ st_5 \leftarrow 15\}$ is consistent with respect to that constraint.

7.3.1 CSP Solving Approach 1

The CSP-standard solving approach for the job-shop consist on traversing depth-first a search tree such that each node corresponds to a partial consistent scheduling of someoperations [Sadeh and Fox, 96]. At each node a new operation is selected and its possible start times are sequentially attempted. Regarding the example of Figure 7.1, this approach searches in a tree of depth 6 (the number of operations). Each node has a different branching factor, depending on the number of possible start times for the current operation.



(a) A job-shop problem



(b) Legend for each operation

Figure 7.1: A job-shop problem.

Under this approach, the variable ordering heuristic selects the next operation to schedule and the value ordering heuristic decides the order in which its possible starting times are tried. We will refer to this approach for job-shop solving as approach 1.

7.3.2 CSP Solving Approach 2

Some authors have reported that the job-shop is more efficiently solved if it is formulated in terms of precedences between pairs of operations competing for the same machine [Smith and Cheng, 93]. Each pair of operations gives two possibilities, the two ways precedence can be established. We denote by $i \rightarrow j$ the fact that operation i precedes operation j. With this approach, search traverses a binary tree. At each search state, it selects a pair of operations competing for the same machine whose precedence has not yet been established, and sequentially attempt the two possible orders (*i.e.*: $i \rightarrow j$ and $j \rightarrow i$). After establishing an order, the intervals of feasible starting times of the two affected operations have to be accordingly updated. If the decision $k \rightarrow l$ is taken, time intervals of operations k and l have to be updated with the following rule:

$$est_l = max\{est_l, est_k + d_k\}$$
$$lft_k = min\{lft_k, lft_l - d_l\}$$

and these new values are then propagated forward or backward respectively through all pre-specified and previously decided precedences. A dead-dend takes place when $est_i + d_i$ becomes greater than lft_i for any operation i. When a dead-end occurs, the algorithm backtracks to a previous decision and changes its precedence relation. We will refer to this approach as CSP solving approach 2.

For instance, in the example of Figure 7.1 there are three pairs of precedences that need to be established. Consequently, the search tree is a binary tree of depth three. If we decide to post operation O_2 before operation O_5, the time interval for O_5 is changed from [14, 25] to [15, 25]. This change is propagated to O_6 and its time interval becomes [20, 26]. Figure 7.2 shows the resulting subproblem after this decision.

Observe that this is not a different CSP representation of the same problem where variables are pairs of operations, because problem constraints are not given in terms of pairs of operations (*i.e.*: the problem description does not explicit whether combinations of precedences are consistent or not). This formulation defines a different solving schema in which search is done in terms of a different type of decisions.

Under this approach there are two decisions that search has to take at each visited node: what is the next pair of operations whose precedence is going to be established, and in what order the two possibilities are

going to be attempted. For parallelism with the solving approach 1, we will refer to this decisions as variable and value selection, respectively.

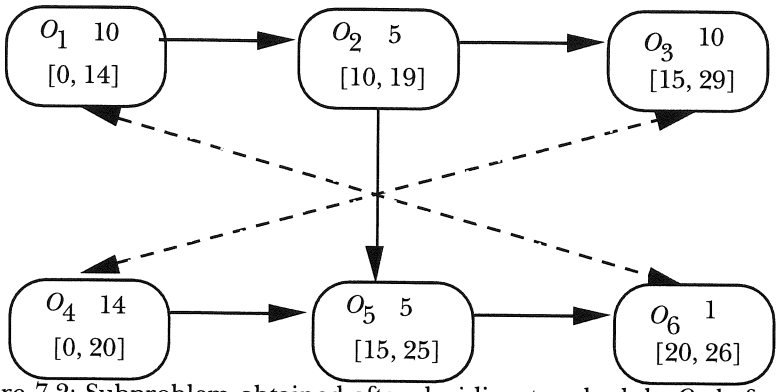


Figure 7.2: Subproblem obtained after deciding to schedule O_2 before O_5 and propagating this decision.

7.4 Support-based Heuristics for the Job-shop

7.4.1 CSP Solving Approach 1

The job-shop that we are considering in this work is a total constraint satisfaction problem. With the CSP solving approach 1, support-based heuristics can be used as they were defined in the previous Chapter. At a given node, there are a set of assigned operations (past operation with a start time assigned to them) and a set of unassigned operations (future operations with a set of feasible values). Assuming a forward-checking-like algorithm, future values are consistent with past assignments. In that context, the support that an arbitrary start time t for an arbitrary operation i receives is,

$$Supp(st_i, t) = |\mathbf{P}| + \sum_{j \in \mathbf{F}} \frac{|Feasible'(D_j)|}{|Feasible(D_j)|}$$

Where $Feasible(D_j)$ is the current domain of operation j and $Feasible'(D_j)$ is the current domain restricted to those values consistent with the assignment of time t to the start time of operation i. The approximate support that an arbitrary start time t for an arbitrary operation i receives is,

$$ap\_Supp(st_i, t) = |\mathbf{P}| + \sum_{j \in \mathbf{F}} \frac{|D'_j|}{|D_j|}$$

Where D_j is the initial domain of operation j and D_j' is the initial domain restricted to those values consistent with the assignment of time t to the start time of operation i.

The lowest support heuristic (LS) selects the variable having the lowest sum of supports among its feasible values. The highest support heuristic (HS) attempts the assignment of values by decreasing support. The approximate counterparts of these heuristics (ALS and AHS) are defined in the same way, but using approximate supports.

7.4.2 CSP Solving Approach 2

With the solving approach 2, nodes are not defined in terms of past and future operations. At a given search node, each operation i has a time interval for its start time ($st_i \in Feasible(D_i)$) determined by previous decisions and the propagation rule. If i and j are two unordered operations competing for the same machine, we define the support that establishing $i \rightarrow j$ receives from the problem as

$$Supp(i \rightarrow j) = \sum_{k \in Operations} \frac{|Feasible'(D_k)|}{|Feasible(D_k)|}$$

where $Feasible(D_k)$ is the time interval $[est_k, lft_k\text{-}d_k]$ for operation k before deciding the order between i and j, and $Feasible'(D_k)$ is the time interval $[est_k, lft_k\text{-}d_k]$ after deciding $i \rightarrow j$ and propagating its effect. With this definition, the lowest support variable selection heuristic (LS) for this algorithm selects the pair of operations (i,j) with minimum sum of supports for the two possible orderings,

$$min_{k,l}\{Supp(k \rightarrow l) + Supp(l \rightarrow k)\}$$

In a similar way, the highest support value selection heuristic (HS) selects the ordering that ordering that receives the highest support.

7.5 Experimental Results

7.5.1 The Benchmark

Several studies of the job-shop from an AI point of view have used the benchmark proposed in [Sadeh, 91] to evaluate their approaches[2]. This fact makes this benchmark especially appropriated for our purposes because using a single set of problem instances we can compare our work with several previous approaches.

The problem set consists of 60 randomly generated problems. Each problem contains 10 jobs and 5 resources. Each job has 5 operations. A controlling parameter was used to generate problems in three different deadline ranges: wide (w), median (m) and tight (t). A second parameter was used to generate problems with both 1 and 2 bottleneck resources. Combining these parameters, 6 different categories of problems were defined and 10 problems were generated for each category. The problem categories were carefully defined to cover a variety of manufacturing scheduling circumstances. All problems have at least one solution. A detailed description of the problem generator can be found in [Sadeh and Fox, 96]. Initial time intervals for operations have around 100 possible starting times, on average. Therefore, using solving approach 1 the associated search space is around 100^{50}. Regarding solving approach 2, there are 225 pairs of operations competing for the same resource. Therefore, the associated search space has size 2^{225}.

7.5.2 CSP Solving Approach 1

The first experiment aimed to show that support-based heuristics produce competitive results when applied to the CSP solving approach 1 using a standard algorithm. We used plain forward checking combined with *support-based* heuristics. The only modification made to the problem description (Section 7.3.1) was that implicit precedences between nonconsecutive operations of the same job were made explicit by additional precedence constraints (in the example of Figure 7.1 it means the addition of two constraints $R_{13}=\{(t,t') \mid t + 15 \leq t'\}$and $R_{46}=\{(t,t') \mid t + 19 \leq t'\}$).

Table 7.1 shows the results of forward checking with support-based heuristics. For each case we give two columns: number of solved problems (with a search limit of 500 visited nodes) and average search effort

[2]This test suite can be obtained via anonymous ftp to cimds3.cimds.ri.cmu.edu (password: user-name@node). The test suite is in /usr/sadeh/public/csp_test_suite.

required as the number of visited nodes. Using exact heuristics (*LS/HS*) 51 problems are solved. If we use the approximate lowest support for variable selection (*ALS/HS*), we still solve the same 51 problems. Interestingly, by approximating support for variable selection average time of solving a problem decreases from about 300 seconds in a Sun workstation to roughly 4 seconds. In addition, all solved problems are solved without any backtracking. If approximate heuristics are used for variable and value selection (*ALS/AHS*), only 39 problems can be solved. It illustrates the importance of value selection heuristic accuracy for scheduling problems. These results are compared with the corresponding ones of [Sadeh and Fox, 96], where the forward checking algorithm with ORR/FSS heuristics solved 52 problems. With their approach, solving a problem required about 8 seconds using a DECstation 5000/200 platform and a Lisp implementation

7.5.3 CSP Solving Approach 2

The second experiment aimed to show that support-based heuristics are also general in the sense that they can be effectively applied to different algorithmic approaches. We use the CSP solving approach 2. In particular, we use the PCP algorithm presented in [Smith and Cheng, 93] combined with support-based heuristics. Unlike [Smith and Cheng, 93], our implementation allows backtracking when a deadend occurs.

Table 7.2 shows the results of this experiment: 56 problems were solved with a search limit of 1,000 nodes. All solved problem instances but one are solved without any backtracking (225 visited nodes). In average, our algorithm requires about 13 seconds to solve a problem. These results are compared to those of [Smith and Cheng, 93] where PCP combined to slack-based heuristics also solved 56 problem. In their approach, 0.2 seconds were required to solve a problem with a C implementation and a DECstation 5000 platform.

7.6 Experimental Results Using Discrepancy Algorithms

An important drawback of depth-first search is that it is strongly committed to its first decisions. If heuristics give a wrong advice early in the tree, depth-first is forced to unsuccessfully traverse a large subtree. In the previous Section, we showed that no heuristic (ORR/FSS, slack-based,

	LS/HS		ALS/HS		ALS/AHS		ORR/FSS	
	solved	nodes	solved	nodes	solved	nodes	solved	nodes
w/1	10	50	10	50	9	55	10	52
w/2	9	50	9	50	8	116	10	50
m/1	9	50	9	50	5	53	8	64
m/2	10	51	10	50	7	71	9	57
t/1	6	50	6	50	4	50	7	68
t/2	7	50	7	50	6	52	8	61
sum	51		51		39		52	

Table 7.1: Results of forward checking with support-based heuristics, compared with results of forward checking with ORR/FSS heuristics of [Sadeh and Fox, 96].

	LS/HS		slack-based	
	solved	nodes	solved	nodes
w/1	10	225	10	225
w/2	10	225	10	225
m/1	10	225	10	225
m/2	10	231	10	225
t/1	10	225	10	225
t/2	6	225	6	225
sum	56		56	

Table 7.2: Results of PCP with support-based heuristics, compared with results of PCP with slack-based heuristics [Smith and Cheng, 93].

support-based) is perfect and, in occasions, it may give wrong advice. In Sadeh's problems, the search space is too large to traverse exhaustively . Therefore, an early mistake causes the failure of the search procedure.

Trying to solve this problem for the job-shop, [Sadeh et al, 95] modified the forward checking algorithm adding an *incomplete backjumping heuristic*: when the system starts thrashing, the algorithm backjumps all the way to the first search state and simply tries the next best value. This approach renders the algorithm incomplete and it has been implemented with a parameter (the maximum number of visited nodes between backjumps), which has to be adjusted to solve the whole benchmark.

Alternatively, [Smith and Cheng, 93] modified the slack-based heuristics introducing a bias; this was implemented by two parameters n_1 and n_2, which should be adjusted manually to achieve the heuristic formulation able to solve the whole benchmark. In both cases, parameters are problem-dependent and they may change using a different benchmark, so manual tuning is always required.

To decrease the degree of dependency of depth-first search with initial decisions in the search tree, new search strategies have been recently proposed following the work of [Harvey and Ginsberg, 95] and further developed by [Korf, 96; Meseguer, 97; Walsh, 97]. These new algorithms are based on the concept of *discrepancy*. Regarding CSP, a search path has as many discrepancies as value assignments differing from the value ordering heuristic first choice. A discrepancy-based algorithm is not strictly committed to the first choices made early in the tree, which forces depth-first to search a sequence of nested subproblems, but it searches in several subtrees corresponding to subproblems which have little in common. This minimizes the negative performance impact of early wrong decisions. In particular, *limited discrepancy search* (LDS, [Harvey and Ginsberg, 95]) is a complete backtracking algorithm that searches the nodes of the tree in increasing order of discrepancies (*i.e.*: in its first iteration it searches all paths with less than 1 discrepancy, in its second iteration it searches all paths with less that 2 discrepancies and so on). LDS is easily adapted to algorithms used in Section 7.5 and combined with support-based heuristics, producing complete procedures which are adaptable to problem difficulty, so no manual tuning of parameters is required. In the following, we provide experimental results of these combinations for the two CSP solving approaches.

7.6.1 CSP Solving Approach 1

Our third experiment aimed to show that the reason for failure in 9 problems of Section 7.5.2. is the combination of two factors: occasional wrong heuristic advice and depth first commitment to early decisions. For this purpose, we combine LDS with forward checking and support-based heuristics (*ALS* and *HS*). Table 7.3 presents the results of the experiment where all problems are solved. 51 problems are solved with 0 discrepancies (and an average CPU time of 4 seconds), 8 problems require 1 discrepancy (and 115 seconds on average) and 1 problem requires 2 discrepancies (and 3,700 seconds). Tracing the execution we could verify our conjecture because solution paths had their discrepancies in the first tree levels (in six cases the discrepancy occurred in the first tree level, in the remaining cases discrepancies occurred in the first four tree levels). As far as we know, this is the first time that a complete algorithm solves all the benchmark problems using the solving approach 1.

7.6.2 CSP Solving Approach 2

Our fourth and last experiment combined LDS with the PCP algorithm and support-based heuristics. Table 7.4 gives the results. Again, all problems are solved: 55 problems with 0 discrepancies (14 seconds on average) and 5 problems with 1 discrepancy (36 seconds on average). All discrepancies occurred in the first four tree levels.

7.7 Conclusions

From the experimental results presented in this Chapter, we conclude that support-based heuristics have in practice the same solving performance than specific heuristics in their initial form using the same problem formulation and the same algorithms (51 vs. 52 solved problems with forward checking, 56 vs. 56 solved problems with PCP). In addition, problem-dependent modifications such as incomplete backjumping heuristic or the inclusion of bias in slack-based heuristics can be efficiently substituted by a discrepancy-based algorithm. This algorithm modifies the depth-first strategy of forward checking and PCP. Combined with support-based heuristics, it reaches the same solving performance (the whole benchmark is solved). In this way, the problem of heuristic mistakes is attacked from a sound algorithmic approach, and ad-hoc modifications and manually adjusted parameters can be avoided.

For all this, we claim that search and CSP techniques, motivated and developed in a generic context, can be effectively applied the job-shop problem. Regarding performance, our generic approach has the same solving power than specific ones; although it does not outperform specific methods in CPU time, its computational requirements are quite reasonable. Regarding methodology, our approach is generic and it does not include domain-dependent elements which have to be adjusted for each problem set. This makes our approach more robust and more applicable to other problem instances. The solution proposed is a combination of three well known elements in the constraint community: constraint propagation (by forward checking), dynamic variable and value selection (by support-based heuristics) and early mistakes avoidance (by discrepancy-based search). By the modular inclusion of each of these elements, we have assessed their relative importance and the role that each plays in the construction of the solution. Each of these elements has an intrinsic value for the search community and it has been independently analyzed and studied. This provides our approach a higher level of understanding than specific methods, which renders it more suitable for supporting the development of applications.

	ALS/HS		inc. fc ORR/FSS	
	solved	nodes	solved	nodes
w/1	10	50	10	52
w/2	10	53	10	50
m/1	10	68	10	55
m/2	10	50	10	54
t/1	10	4,831	10	57
t/2	10	813	10	60
sum	60		60	

Table 7.3: Results of LDS with forward checking and support-based heuristics, compared with results of incomplete forward checking with ORR/FSS heuristics [Sadeh et al, 95].

	LS/HS		modif. slack-based	
	solved	nodes	solved	nodes
w/1	10	225	10	225
w/2	10	225	10	225
m/1	10	225	10	225
m/2	10	244	10	225
t/1	10	225	10	225
t/2	10	356	10	225
sum	60		60	

Table 7.4: Results of LDS with PCP and support-based heuristics, compared with results of PCP with modified slack-based heuristics [Smith and Cheng, 93].

Chapter 8

Conclusions

Many important problems can be expressed as CSP. The development of techniques to efficiently solve them is of clear practical importance. In this thesis, we have presented a set of algorithms which improve state-of-the-art methods for constraint satisfaction solving. We have shown that total and partial constraint satisfaction have many common features that can be exploited. In that context, we have extended many technics developed for total constraint satisfaction to partial constraint satisfaction. Moreover, we have developed some techniques that have been shown to be useful in both cases. All our research has been motivated under a general-purpose perspective, without assuming any domain knowledge. For this reason, we believe that our contributions can be effective in a broad spectrum of domains.

Our work has some recognized limitations. For instance, we have only considered binary problems. Altough most work related to ours also assumes binary problems, it has to be mentioned that many practical problems have their natural representation as a n-ary CSP. Part of our claims on the efficiency of our algorithms is supported by an experimental evaluation based on random problems. It should be clear that results obtained on random problems need not extrapolate to every particular domain. Finally, we have limited our research to systematic algorithms. Therefore, we have disregarded stochastic search methods, an important line of research which has been very fruitful in constraint satisfaction.

This thesis leaves a number of doors open. Some of our algorithms can be more efficiently implemented. Some of our techniques can be naturally combined, and some of our ideas can be extended to more general constraint satisfaction frameworks.

Considering the intractability of constraint satisfaction problems, the aim of our research is to be a contribution to the development of new algorithms of increasing efficiency which will eventually allow for the successful application of constraint technology to a broad spectrum of problems.

8.1 Conclusions

From our work, we can extract the following general conclusions.

1. An idea that has been at the heart of our work is that one can successfully exploit common features of total and partial constraint satisfaction. The practical importance of this idea is twofold: on the one hand, techniques that have been developed in total constraint satisfaction can be adapted to partial constraint satisfaction. On the other hand, it is possible to develop general techniques and incorporate them into algorithms for both total and partial constraint satisfaction. Throughout this work, several examples of this claim have been presented.
   a. *Subproblem merging*: when a CSP has different values with the same constraining behaviour, standard depth-first search produces similar subproblems and solves them independently, without taking any advantage of their similarity. We have shown that this situation occurs in both total and partial constraint satisfaction and we have proposed a solution which has shown to be effective in both cases. Our approach is based on a search space transformation such that search trees of similar subproblems are merged into a unique tree. The cost is the addition of extra tree levels. We have embedded this idea into forward checking and have developed two algorithms (FCw and PFCw)
   b. *Combining search with local consistency*: when depth-first search falls into a dead-end, a subtree has to be unsuccessfully traversed. During its traversal, all paths are condemned to a dead-end detection. The earlier the dead-end is detected, the sooner search will break out of the dead-end. One of the most fruitful approaches for early dead-end detection in total constraint satisfaction involves the combination of search and local consistency. We have shown that the same idea can be extended to partial constraint satisfaction. Our approach is based on the use of DAC to improve the branch and bound lower bound. We have shown that this approach can produce exponentially large gains with respect to algorithms not using DAC information. We have developed a set of algorithms substantiated on that idea (PFC-GDAC, PFC-RDAC and PFC-MDAC) which have shown to be very effective.
   c. *Lazy evaluation*: look-ahead algorithms propagate their assignments toward future variables. It is a matter of fact that propagation plays an important role in early dead-end detection. In general, stronger propagations have better dead-end capabilities at the cost of performing more computation at each node. Therefore, there is a trade off between the amount

of propagation and the gains that may come with it. In that context, it is of clear interest the development of efficient propagation techniques. It has been observed that some propagation procedures in total constraint satisfaction perform more work than estrictly needed. This lack of efficiency has been successfully solved applying lazy evaluation techniques. We have shown that algorithms for partial constraint satisfaction also suffer from this inefficiency and we have shown that a lazy approach is very appropriate for it. Our approach is based on the application of lazy techniques in order to perform the minimum amount of computation to compute the dead-end detection condition. As a result, we have developed an algorithm (PLFC) which never performs worse than its greedy counterpart and which has the potential to perform much better.

   d. *Heuristics*: We have analyzed the role that heuristics play in algorithms for total and partial constraint satisfaction and we have shown that, although not being equivalent situations, it is sensible to develop heuristics for both problems following the same principles. From an analysis of the common aspects between LP and CSP, we have obtained a unifying view of constraint satisfaction in terms of global optimization. Using this optimization perspective as a source of inspiration, we have developed a variable and a value ordering heuristic (LS and HS) to guide depth-first search. An important feature of these heuristics is that they can be used with algorithms for both total and partial constraint satisfaction.

2. An important feature common to all our work is that it has been developed under a general purpose motivation. Therefore, all our algorithms are general and we believe that they can be specialized to particular domains without jeopardizing their performance. We have presented a good example of this claim in the job-shop context. Our heuristics do not encode any domain specific knowledge and they have shown to be competitive with specialized methods. In that sense, our work supports the importance of developing generic algorithms which in general are easier to develop and maintain, have a broader applicability and can be a starting point for the development of specialized techniques.
3. This thesis has been concerned with systematic algorithms. One may believe that systematicity is a useless feature when dealing with exponential worst-case algorithms. Our work contradicts this belief to some extent. We have shown that different exponential worst-case algorithms may show a completely different behaviour with the same problem instance (*i.e.:* the same problem can be exponentially difficult for one algorithm and trivial for another). In our work, we have improved the efficiency of existing algorithms (some times the

improvement rate has been of several orders of magnitud). Thus, our algorithms contribute to the successful applicability of constraint satisfaction techniques to larger and more difficult instances.

4. The use of arc-consistency information for lower bound computation has lead us to the discovery of a complexity peak on MAX-CSP. We have shown that the search effort of branch and bound enhanced with the use of directed arc consistency counts presents an easy-hard-easy pattern in the average difficulty when solving random binary CSP instances. This discovery generalizes the complexity peak observed (and widely studied) in the decision problem to the optimization case. In MAX-CSP, this phenomenon cannot be explained in terms of a phase transition on the problems solvability. We have given an algorithmic dependent explanation in terms of lower and upper bounds of the branch and bound algorithm. A similar phenomenon has been already detected in other optimization problems such as the travelling salesman problem [Cheeseman *et al.*, 91].

8.2 Further Research

This work raises a number of issues that require further research. We have identified the following,

- Regarding subproblem merging, we gave a very vague notion of value similarity. We believe that our vague notion of value similarity can be characterized using some distance measurement (*i.e.*: taking the set of supporting values and considering common occurrences). If an effective distance measurement can be found and it can be computed in an efficient way, the decision of what values are weakly assigned can be done in a domain independent and automatic way.
- Regarding the combination of search with local consistency enforcement, we believe that two of our algorithms can be further improved. It is our belief that lower bound quality is the major issue in branch and bound algorithms for partial constraint satisfaction. Considering the simplicity of the greedy optimization algorithm presented in Section 4.7, we think that there is still room for further improvements. Regarding the algorithm that maintains RDAC updated, we believe that it can also be improved by moving to more advanced AC schemas. Our research has been restricted to the use of DAC. It may be fruitful to look at other forms of local consistency focusing on the detection of new inconsistencies. The MAX-CSP complexity peak that we have discovered also deserves further study. A deeper

analysis aiming at a good characterization of the peak is of obvious interest.

- Regarding our support-based heuristics, we identify a number of issues which require additional work. Since constraint satisfaction can be seen as the global optimization of the average local consistency function, it is of obvious interest to study this function topology. For instance, it would be interesting to know what happens to this function on random problems as they approach the tightness critical point. A second idea for future work is the extention of our approach to stochastic search algorithms, where heuristic guidance cannot be explained in terms of the fail-first and succeed-first principles.

- An important feature of CSP techniques is that they can be easily combined. Throughout our work, we have shown some examples of how our different algorithms could be combined (PFCw-DAC in Chapter 3, PLFC-DAC in Chapter 5, PFC-RDAC plus support-based heuristics in Chapter 6). However, most of these hybrid algorithms were basic versions. There are some other combinations that seem to be especially promising. We have identified the following two:

  PLFC-MRDAC is the algorithm that maintains reversible DAC updated in a lazy manner. It seems a suitable combination because since maintaining DAC is expensive, a lazy approach has more to save.

  FCw plus support-based heuristics seem to be a fruitful combination because computing the support of two values (required for the heuristics) and evaluating their common constraining behaviour (useful for weak assignments) can be done simultaneously. Therefore, it gives two different ways in which to take advantage of the overhead.

Appendix

Solving Fuzzy Constraint Satisfaction Problems

In this thesis, we have associated partial constraint satisfaction with MAX-CSP, where every constraint is equally important and only give two possible constraining values: *totally allowed* or *totally disallowed*. In the last years, new types of constraints have found to be useful to model realistic problems, allowing for intermediate satisfaction degrees. Some examples are flexible constraints, priority constraints and conditional constraints. Fuzzy set theory seems to be specially well suited to model these new types of constraints in an integrated form. This has generated a new kind of problem denominated Fuzzy CSP (FCSP), as an extension of classical CSP where constraints are represented by fuzzy relations. In this case, a solution is a complete assignment which satisfies, at least partially, every constraint and with maximum satisfaction degree on the whole set of constraints. FCSP is an optimization problem, similar to MAX-CSP.

In this Appendix, we extend part of our results presented in Chapter 4 to FCSP. More precisely we present the generalization of DAC usage to the fuzzy case. We have tested our algorithms on random problems, showing empirically how lower bound improvement increases algorithm efficiency on average.

This Appendix is organized as follows. In Section A.1, we extend the classical CSP framework to the fuzzy case. In Section A.2, we define branch and bound search for FCSP solving, introducing different lower bounds. In Section A.3, we present empirical results of testing our algorithms on random problems. Finally, in Section A.4 we summarize this work.

A.1 Extending Classical CSP

In classical CSPs constraints are required to be crisp (i.e., a pair of values either satisfies or violates completely a constraint). This model can be

generalized allowing the introduction of flexible constraints, defined as follows. A *flexible* binary constraint R_{ij} is an application from $D_i \times D_j$ into a completely ordered set $(E, >)$ [Schiex *et al.*, 95], where the maximum element stands for complete satisfaction, the minimum element stands for complete violation, and intermediate elements represent partial degrees of satisfaction. A flexible CSP is a classical CSP where crisp constraints are substituted by flexible constraints. A solution is a complete assignment satisfying, at least partially, every constraint with the maximum satisfaction degree on the whole set of constraints, where some operator is used to combine the individual satisfaction degree of each constraint.

In classical CSPs all constraints are equally important. This is not always the case in real problems, for which constraints with priority have been introduced. Given a CSP with crisp constraints, the priority of a constraint $Pr(R_{ij})$ expresses the importance of R_{ij} to be satisfied. Priority is defined as a mapping $Pr:\{R_{ij}\} \rightarrow [0,1]$, where $Pr(R_{ij}) = 1$ means that R_{ij} must be necessarily satisfied and $Pr(R_{ij}) = 0$ means that R_{ij} can be ignored. For intermediate values of priority, $Pr(R_{kl}) > Pr(R_{ij})$ means that if R_{kl} and R_{ij} cannot be satisfied simultaneously, satisfaction of R_{kl} is preferred over satisfaction of R_{ij}.

A.1.1 Fuzzy Modelling of Constraints

Fuzzy set theory can be used to model flexible and priority constraints [Fargière, 94]. A flexible constraint R_{ij} is represented by a fuzzy relation C_{ij} , defined by,

$$\mu_{C_{ij}}(a,b): D_i \times D_j \rightarrow [0,1], \text{ s.t. } \mu_{C_{ij}}(a,b) = R_{ij}(a,b)$$

where set $(E,>)$ corresponds to the set $[0,1]$ with the usual total order in real numbers. In possibility theory, the priority of a constraint $Pr(R_{ij})$ can be interpreted as the necessity of R_{ij} to be satisfied, while $1 - Pr(R_{ij})$ is the possibility of R_{ij} to be violated. This interpretation induces a fuzzy relation C_{ij} in the set $D_i \times D_j$ defined by,

$$\mu_{C_{ij}}(a,b) = 1, \qquad \text{if } (a,b) \text{ satisfies } R_{ij}$$
$$\mu_{C_{ij}}(a,b) = 1 - Pr(R_{ij}) \qquad \text{if } (a,b) \text{ does not satisfy } R_{ij}$$

Although more sophisticated constraint types can also be formulated in this model, in this Appendix we will focus on flexible and priority constraints.

A.1.2 Fuzzy CSP

A *fuzzy* CSP (FCSP) is defined by a set of n variables $\{X_i\}$ taking values on discrete and finite domains $\{D_i\}$ under a set of e constraints $\{R_{ij}\}$ represented as fuzzy relations $\{C_{ij}\}$. A *solution* is a complete assignment which satisfies, at least partially, every constraint and with the maximum satisfaction degree.

The satisfaction degree of a global assignment $A=\{X_i \leftarrow v^i: 0 \leq i \leq n\}$ on the whole set of constraints is given by the intersection of the fuzzy relations corresponding to every constraint, defined by,

$$\min_{ij} \{\mu_{C_{ij}}(v^i, v^j)\} \qquad \text{(A.1)}$$

A solution of the FCSP is the complete assignment with maximum satisfaction degree of the least satisfied constraint,

$$\max_A \{\min_{ij} \{\mu_{C_{ij}}(v^i, v^j)\}\} \qquad \text{(A.2)}$$

where $A \in D_1 \times \ldots \times D_n$. This analysis can be repeated substituting the satisfaction degree of a constraint, $\mu_{C_{ij}}$, by its *violation degree*, defined as $1 - \mu_{C_{ij}}$. Then, a solution is the complete assignment with minimum violation degree of the most violated constraint,

$$\min_A \{\max_{ij} \{1 - \mu_{C_{ij}}(v^i, v^j)\}\}$$

Both equations (A.1) and (A.2) are equivalent. In the following, we will use equation (A.2) for FCSP solving.

A.1.3 The Lexicographic Approach

Equation (A.2) causes a too coarse solution generation because it only considers the most violated constraint. To refine this, we take the *lexicographic* approach [Dubois *et. al.*, 96], which discriminates among assignments sharing the violation degree of the most violated constraint. The compatibility of a partial assignment $A=\{X_i \leftarrow v^i : X_i \in V\}$ with respect to a subset V of the problem constraints is described by its vector of *violation degrees*, $VD(A)$, defined as follows,

$$[VD(A)]_{ij} = 1 - \mu_{C_{ij}}(v^i, v^j), \qquad \text{if } X_i, X_j \in V$$
$$[VD(A)]_{ij} = 0, \qquad \text{otherwise}$$

$VD(A)$ has e components, one for each constraint. Component (i,j) contains the violation degree of R_{ij} by the assignment A. Vectors of violation degrees are ranked by increasing lexicographic ordering. Given two assignments A and B such that $VD(A) = (a_1, \ldots, a_e)$ and $VD(B) = (b_1, \ldots, b_e)$, its lexicographical ordering is defined as follows: (i) rearrange vectors in

decreasing order, $a_{i_1} \geq ... \geq a_{i_e}$ and $b_i \geq ... \geq b_{i_e}$, and (ii) perform a lexicographical comparison starting from the leftmost component, $A <_{lex} B \Leftrightarrow \exists\, k \leq e$ such that $\forall\, l<k$, $a_{il} = b_{il}$ and $a_{ik} < b_{ik}$.

The preferred solution is the assignment with minimum vector of violation degrees. With this formulation, equation (A.2) is replaced by,

$$\min^{lex}_{A} \{\max_{ij} \{1 - \mu_{C_{ij}}(v^i, v^j)\}\} \qquad \forall A \in D_1 \times ... \times D_n$$

A.2 FCSP and Branch and Bound

Finding a solution to a FCSP is an optimization problem, where the goal is to find the complete assignment with minimum vector of violation degrees satisfying, at least partially, every constraint. For this kind of problems we can also use depth-first branch and bound. In the following, we assume that variables are instanciated following a predetermined sequence $\{X_1, X_2, ..., X_n\}$. At an arbitrary node, we have the set of past variables **P**, the set of future variables **F**, the best solution found in the explored part of the tree *UB* (the upper bound) and an underestimation of the best solution in the current subtree *LB* (the lower bound). Each node has an associated partial assignment $A=\{X_i \leftarrow v^i : X_i \in \mathbf{P}\}$.

In the FCSP case, the cost function is given by the vector of violation degrees with $<_{lex}$ order. Therefore, the vector of the current best solution, VD_{bs}, is taken as the upper bound, $UB = VD_{bs}$. For the lower bound, a first option is to take the vector of the current node, VD_{cur}, given that no node in the subtree rooted at the current node can have a vector lower than VD_{cur}. In the following subsections, we develop strategies for more sophisticated lower bounds.

Pruning occurs when any of the following conditions occurs at the current node, (i) a constraint is violated completely or (ii) $LB \geq_{lex} UB$. Regarding (i), if the assignment of the current node violates completely a constraint, any extension of this assignment will violate the same constraint. Therefore, the subtree rooted at this node contains no solutions and it can be pruned. Regarding (ii), $LB \geq_{lex} UB$ is the standard pruning condition of branch and bound adapted to our case. It means that any leaf descending from the current node will have a vector of violation degrees greater than or equal to *UB* in the lexicographical order, so there is no point in visiting them and the subtree rooted at the current node can be pruned. When the current node is a leaf where $VD_{cur} <_{lex} VD_{bs}$, the assignment of the current node is better than the current best solution. This assignment becomes the new current best solution and the *UB* is updated accordingly. The solution of the problem is the current best solution after branch and bound traverses the whole search tree.

A.2.1 Lower Bound Approaches

At a given node, the lower bound is an underestimation of the minimum value of the cost function among leaves descending from it. Regarding FCSP, the lower bound is a vector of violation degrees lower than or equal to the minimum vector among descending leaves. We define three sets of constraints at the current node,

$$R^{PP} = \{R_{ij}: X_i, X_j \in \mathbf{P}\}$$
$$R^{PF}(X_i) = \{R_{ij}: X_j \in \mathbf{P}, X_i \in \mathbf{F}\}$$
$$R^{FF}(X_i) = \{R_{ij}: X_i, X_j \in \mathbf{F} \text{ and } i<j\}$$

The simplest lower bound comes from considering constraints in R^{PP} only. If A is the current partial assignment, it is obvious that any descending leaf will violate constraints in R^{PP} with the same degree that A does. Therefore, we define our first lower bound, LB_1,

$$[LB_1]_{ij} = [VD(A)]_{ij}, \quad \text{if } R_{ij} \in R^{PP}$$
$$[LB_1]_{ij} = 0, \quad \text{otherwise}$$

where non-zero components in LB_1 correspond to constraints in R^{PP}. This lower bound extends the idea of using the *distance* as lower bound to the fuzzy case. A better lower bound can be computed taking advantadge of the effect that the assignment A has on future variables, or equivalently, performing lookahead on future variables. The effect that A has on a value a of a future variable X_i is represented by its look-ahead vector of violation degrees, $VD^L(X_i, a)$,

$$[VD^L(X_i, a)]_{ij} = 1 - \mu_{C_{ij}}(a, v^j), \quad \text{if } R_{ij} \in R^{PF}(X_i)$$
$$[VD^L(X_i, a)]_{ij} = 0, \quad \text{otherwise}$$

which plays the same role as inconsistency counts in PFC for MAX-CSP.

Every leaf descending from the current node will have some value assigned to every future variable. Any assignment of X_i will contribute to violate constraints in $R^{PF}(X_i)$ to a degree which will be, at least, the minimum $VD^L(X_i, a)$ for all $a \in D_i$. Denoting this value by a_{min_L} and applying this idea to every future variable, we obtain a second lower bound LB_2 ,

$$[LB_2]_{ij} = [VD(A)]_{ij}, \quad \text{if } R_{ij} \in R^{PP}$$
$$[LB_2]_{ij} = [VD^L(X_i, a_{min\_L})]_{ij}, \quad \text{if } R_{ij} \in R^{PF}(X_i)$$
$$[LB_2]_{ij} = 0, \quad \text{otherwise}$$

which extends to the fuzzy case the lower bound described in Section 2.3.

The lower bound can still be refined performing some level of directional arc-consistency among future variables. Regarding FCSP, given

two future variables X_i and X_j ($i<j$), constraining each other by $R_{ij} \in R^{FF}(X_i)$, if value $a \in D_i$ is assigned to X_i , the minimum violation degree for R_{ij} will be $\min_b \{(1 - \mu_{C_{ij}}(a, b))\}$, for all $b \in D_j$. We can combine this effect on X_i from future variables appearing after it in the instantiation order, with the effect caused from past variables on X_i , in the extended lookahead vector of violation degrees, $VD^{EL}(X_i, a)$,

$$
\begin{aligned}
&[VD^{EL}(X_i, a)]_{ij} = 1 - \mu_{C_{ij}}(a, v^i), && \text{if } R_{ij} \in R^{PF}(X_i)\\
&[VD^{EL}(X_i, a)]_{ij} = \min_b \{ (1 - \mu_{C_{ij}}(a, b))\}, && \text{if } R_{ij} \in R^{FF}(X_i)\\
&[VD^{EL}(X_i, a)]_{ij} = 0, && \text{otherwise}
\end{aligned}
$$

Any assignment of X_i will contribute to violate constraints in $R^{PF}(X_i) \cup R^{FF}(X_i)$ to a degree that will be, at least, the minimum $VD^{EL}(X_i, a)$ for all $a \in D_i$. Denoting this value by a_{min_EL} and applying this idea to every future variable, we obtain a third lower bound LB_3,

$$
\begin{aligned}
&[LB_3]_{ij} = [VD(A)]_{ij}, && \text{if } R_{ij} \in R^{PP}\\
&[LB_3]_{ij} = [VD^{EL}(X_i, a_{min\_EL})]_{ij}, && \text{if } R_{ij} \in R^{PF}(X_i) \cup R^{FF}(X_i)
\end{aligned}
$$

This is the extension of the lower bound described in Section 4.4 to the fuzzy case.

It is important to notice that $LB_3 \geq_{lex} LB_2 \geq_{lex} LB_1$. Although LB_3 is the best lower bound among the three formulations, the selection of a specific bound is a matter of balance between its computational cost and the benefit it causes (in terms of pruning efficiency), for a class of problems.

A.2.2 Pruning Domain Values

Branch and bound can also prune values on domains of future variables, when using LB_2 or LB_3. Considering LB_3, pruning conditions for domain values can be developed as follows. If $X_i \in \mathbf{F}$ and $a \in D_i$, we can prune value a at the current node providing,

$$LB_3 - VD^{EL}(X_i, a_{min\_EL}) + VD^{EL}(X_i, a) \geq_{lex} UB$$

where + and - represent vector addition and subtraction. The rationale for this condition is as follows. $VD^{EL}(X_i, a_{min\_EL})$ is the minimum contribution of any assignment of X_i regarding violation of constraints in $R^{PF}(X_i) \cup R^{FF}(X_i)$. $VD^{EL}(X_i, a)$ is the actual contribution of the assignment (X_i, a) regarding violation of the same constraints. If we substitute the minimum contribution of X_i to LB_3 by the contribution of the assignment (X_i, a), and the resulting vector is greater than or equal to UB, value a can be pruned because it will never appear as value for X_i in a complete assignment extending the current one and better than the

current best solution. Value pruning depends on the current assignment A, so when branch and bound performs backtracking behind A, every value pruned when A was at the current node should be restored.

A.3 Experimental Results

We have tested the branch and bound algorithm with LB_1, LB_2 and LB_3 on random problems with binary constraints. To do this, we have extended the four parameter model to include FCSP with flexible and priority constraints. A random FCSP with flexible constraints is defined by $<n, m, p_1, p_2^{cv}, p_2^{pv}, g>$ where n, m and p_1 are defined as in the classical case. p_2^{cv} is the ratio of tuples causing complete violation at each constraint (exactly $p_2^{cv}\ m^2$ tuples), and p_2^{pv} is the ratio of tuples causing partial violation at each constraint (exactly $p_2^{pv}\ m^2$ tuples). For these problems, constraint tightness is $p_2 = p_2^{cv} + p_2^{pv}$. Finally, g is a finite number of different satisfaction degrees.

A random FCSP with priority constraints is defined by $<n, m, p_1^{Pr=1}, p_2^{cv}, p_1^{Pr<1}, p_2^{pv}, g>$ where n, m and g are defined as before. $p_1^{Pr=1}$ is the problem connectivity for constraints with priority 1 (the problem has exactly $p_1^{Pr=1}\ n\ (n\text{-}1)/2$ constraints of this type). p_2^{cv} is the tightness of constraints with priority 1 (exactly $p_2^{cv}\ m^2$ forbidden tuples per constraint). $p_1^{Pr<1}$ is the connectivity for constraints with priority lower than 1 (the problem has exactly $p_1^{Pr<1}\ n\ (n\text{-}1)/2$ constraints of this type) p_2^{pv} is the tightness of constraints with priority lower than 1 (exactly $p_2^{pv}\ m^2$ forbidden tuples per constraint). For these problems, problem connectivity is $p_1 = p_1^{Pr=1} + p_1^{Pr<1}$, while two different tightness are present, p_2^{cv} for mandatory constraints and p_2^{pv} for non-mandatory ones.

To construct a random instance, constraints and tuples are sequentially selected following a uniform probabilistic distribution until the correct amount is reached. Experiments on random instances of classical CSP show that, when tightness increases, problems suddently become over-constrained after a critical point. In our models of random FCSP we differentiate tuples causing complete violation from tuples causing partial violation, in order to prevent that tuples causing complete violation render random instances unsoluble, in which the effect of fuzzy constraints is null. Our benchmark comprises twelve sets of problems:

(1). $<30, 3, 100/435, 2/9, p_2^{pv}, 5>$
(2). $<30, 3, 120/435, 1/9, p_2^{pv}, 5>$
(3). $<30, 3, 150/435, 1/9, p_2^{pv}, 5>$
(4). $<30, 3, 100/435, 0, p_2^{pv}, 5>$
(7). $<30, 3, 100/435, 2{,}20/435, p_2^{pv}, 5>$
(8). $<30, 3, 120/435, 1, 20/435, p_2^{pv}, 5>$
(9). $<30, 3, 150/435, 1, 20/435, p_2^{pv}, 5>$
(10). $<30, 3, 0, 0, 100/435, p_2^{pv}, 5>$

(5). <30, 3, 120/435, 0, p_2^{pv}, 5>
(6). <30, 3, 150/435, 0, p_2^{pv}, 5>
(11). <30, 3, 0, 0, 120/435, p_2^{pv}, 5>
(12). <30, 3, 0, 0, 150/435, p_2^{pv}, 5>

For each set of problems p_2^{pv} was varied from its minimum to its maximum value in steps of 1/9. Observe that the benchmark can be classified into four groups: problems with flexible constraints and total violation tuples (1,2,3), problems with flexible constraints without total violation tuples (4,5,6), problems with priority constraints having mandatory constraints (7,8,9) and problems with priority constraints without mandatory constraints (10,11,12). In those sets of problems with tuples causing complete violation, their parameters were selected as to have the highest p_2^{cv} before problems become unsoluble. For each parameter setting 20 instances were generated.

When using LB_1 and LB_2, variables were dynamically ordered by increasing remaining domain size. When using LB_3, variables were statically ordered by decreasing degree. Values were always considered in lexicographical order. Regarding implementation quality, all experiments share the same branch and bound implementation (with the minimal differences associated to the three bounds). When using LB_3, a preprocessing is required to initialize VD^{EL}. Consistency checks performed in this preprocessing are included in the results.

Figures A.1-A.4 shows the results of our experiments. Each figure includes results of a group of three different classes of problems. Each plot corresponds to a set of problems. Plots represent the average number of consistency checks versus the number of tuples causing partial violation in a constraint (the number of visited nodes and CPU time give very similar results). It can be observed that problems without tuples causing complete violation (shown in Figure A.2) are harder to solve. The reason is that tuples causing complete violation have an inmediate pruning effect on values of future variable and reduce the search space. The second observation is that LB_1 is clearly inefficient. LB_2 outperforms LB_1 in all cases and LB_3 outperforms LB_1 in all but two sets (1) and (7).

Comparing LB_2 versus LB_3 , a deeper analysis is required. Regarding sets (1),(2) and (3) (shown in figure A.1), LB_2 equals or dominates LB_3 in the whole range of p_2^{pv}. Something similar occurs in sets (7), (8) and (9) (Figure A.3) for low and medium values of p_2^{pv}, while for high values of p_2^{pv} LB_2 increases abruptly while LB_3 drops gracefully to zero. Regarding sets (4), (5), (6), (10), (11) and (12) (shown in Figures A.2 and A.4), LB_3 outperforms clearly LB_2 . From these results we can conclude that LB_2 is competitive with LB_3 in problems where contraints are completely violated by some value tuple or with mandatory constraints, while LB_3 dominates in problems where constraints cannot be completely violated or without mandatory constraints. Regarding FCSP with priority constraints, LB_3 produces a change in the difficulty pattern of problems like the one detected in Chapter 4. If LB_2 is used, problems become harder

as p_2^{pv} is increased. However, the use of LB_3 produces an easy-hard-easy pattern, where the tightest problems become trivial.

A.4 Conclusions

From this work we can extract the following conclusions. First, algorithmic approaches developed for MAX-CSP can be successfully adapted and applied to FCSP. Second, regarding branch and bound lower bounds, lookahead from past to future variables seems to be always relevant, while lookahead from future into future variables produces relevant savings on problems where constraints cannot be completely violated or without mandatory constraints. And third, problems with priority constraints show an easy-hard-easy pattern in the search effort similar to that observed in Chapter 4.

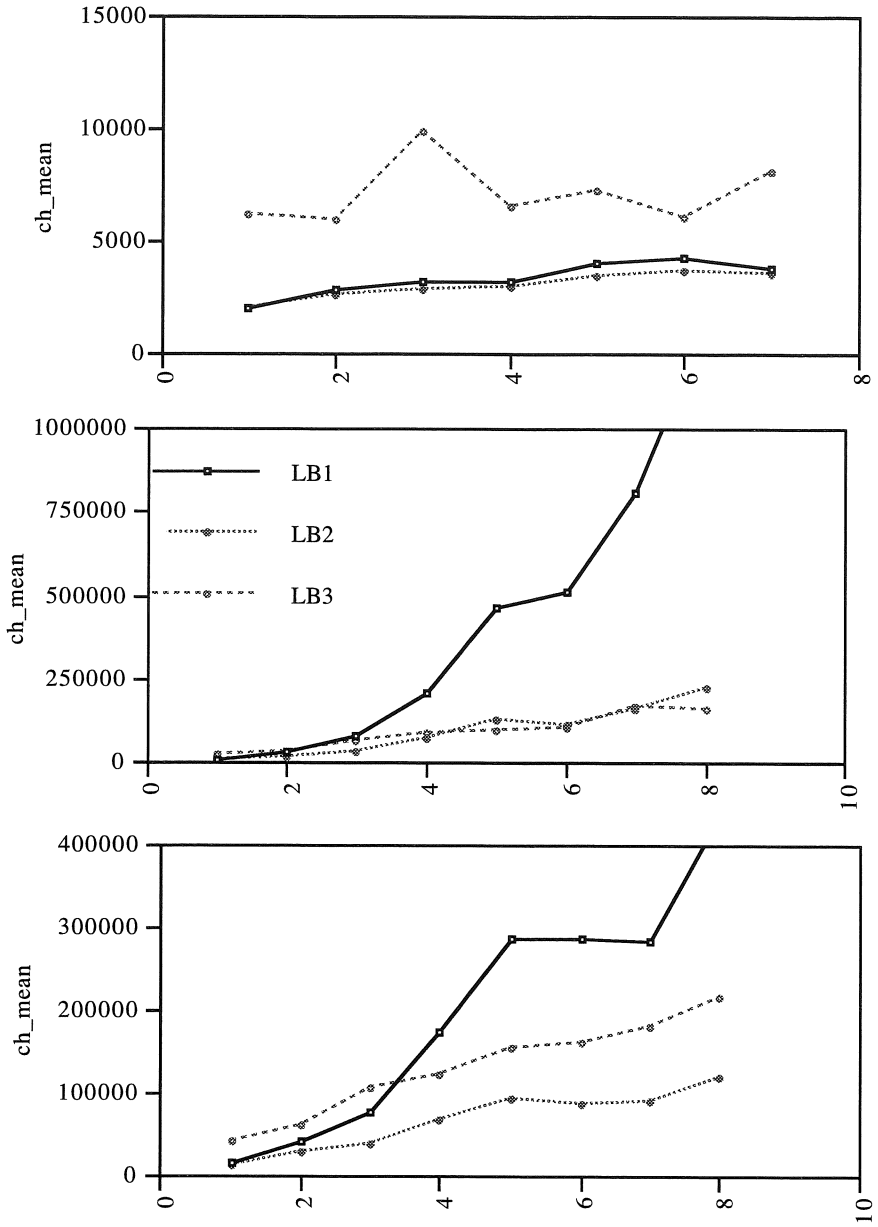


Figure A.1: Experimental results of branch and bound using three different lower bounds on the <30, 3, 100/435, 2/9, $p_2{}^{pv}$, 5>, <30, 3, 120/435, 1/9, $p_2{}^{pv}$, 5> and <30, 3, 150/435, 1/9, $p_2{}^{pv}$, 5> classes of random problems (from top to bottom). Horizontal axis represents $p_2{}^{pv}$ x 9. Vertical axis stands for average number of consistency checks.

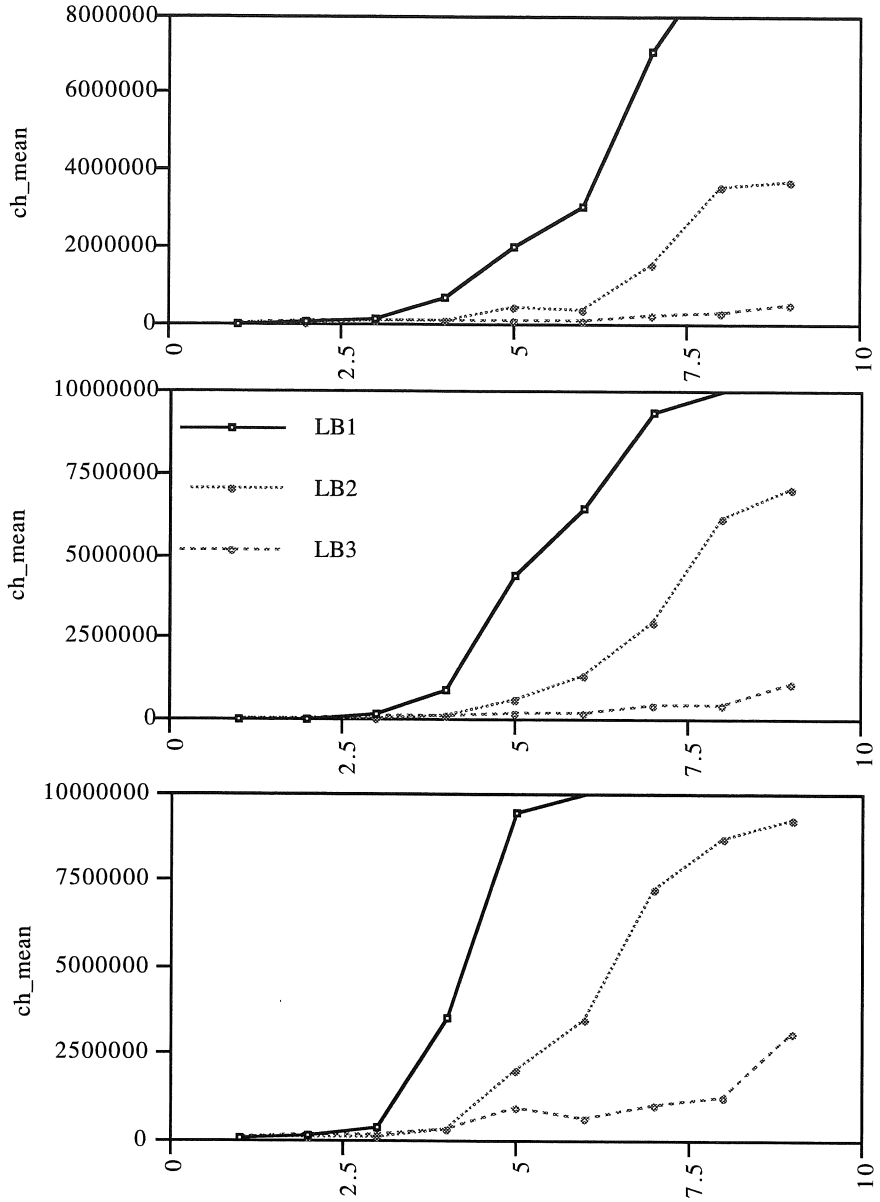


Figure A.2: Experimental results of branch and bound using three different lower bounds on the <30, 3, 100/435, 0, $p_2{}^{pv}$, 5>, <30, 3, 120/435, 0, $p_2{}^{pv}$, 5> and <30, 3, 150/435, 0, $p_2{}^{pv}$, 5> classes of random problems (from top to bottom). Horizontal axis represents $p_2{}^{pv}$ x 9. Vertical axis stands for average number of consistency checks.

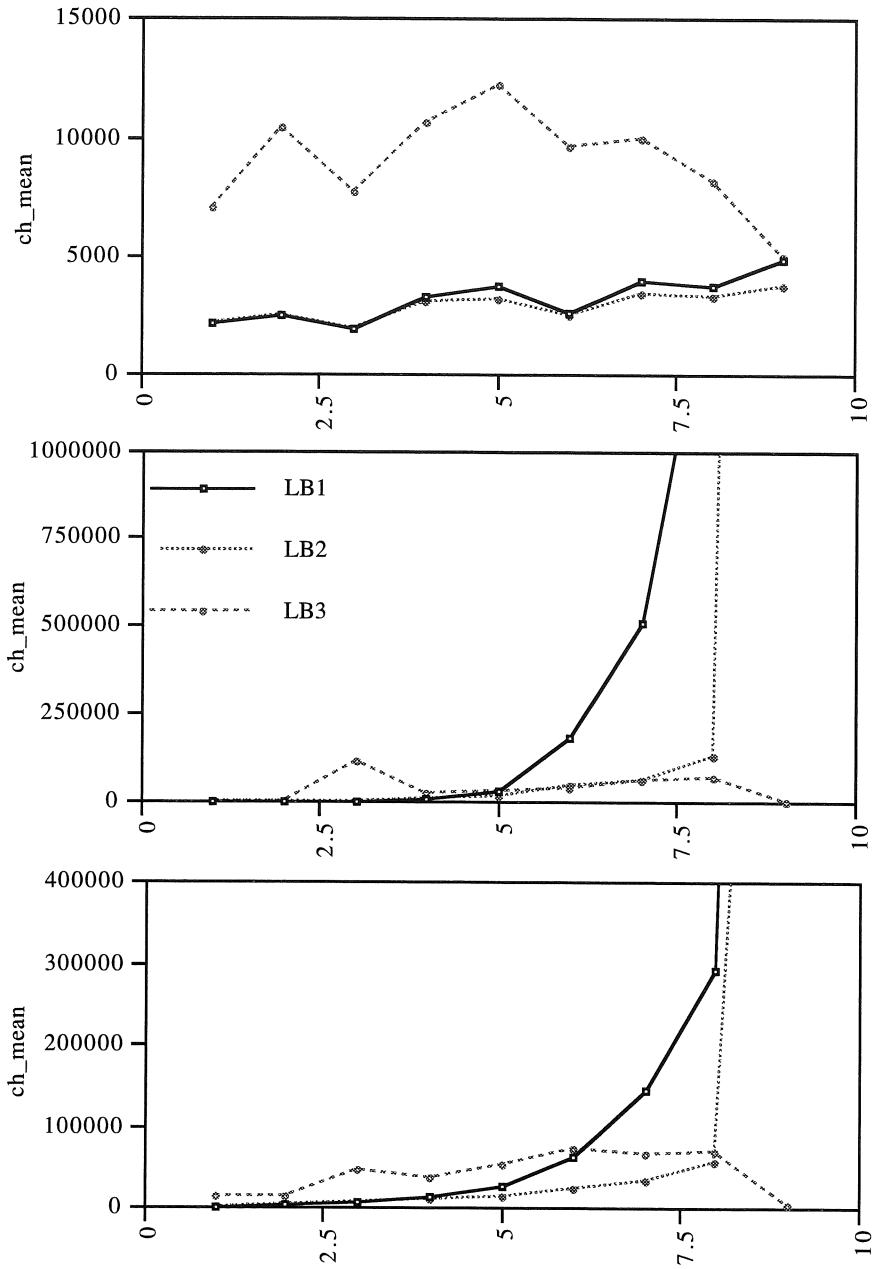


Figure A.3: Experimental results of branch and bound using three different lower bounds on the <30, 3, 100/435, 2, 20/435, $p_2{}^{pv}$, 5>, <30, 3, 120/435, 1, 20/435, $p_2{}^{pv}$, 5> and <30, 3, 150/435, 1, 20/435, $p_2{}^{pv}$, 5> classes of random problems (from top to bottom). Horizontal axis represents $p_2{}^{pv}$ x 9. Vertical axis stands for average number of consistency checks.

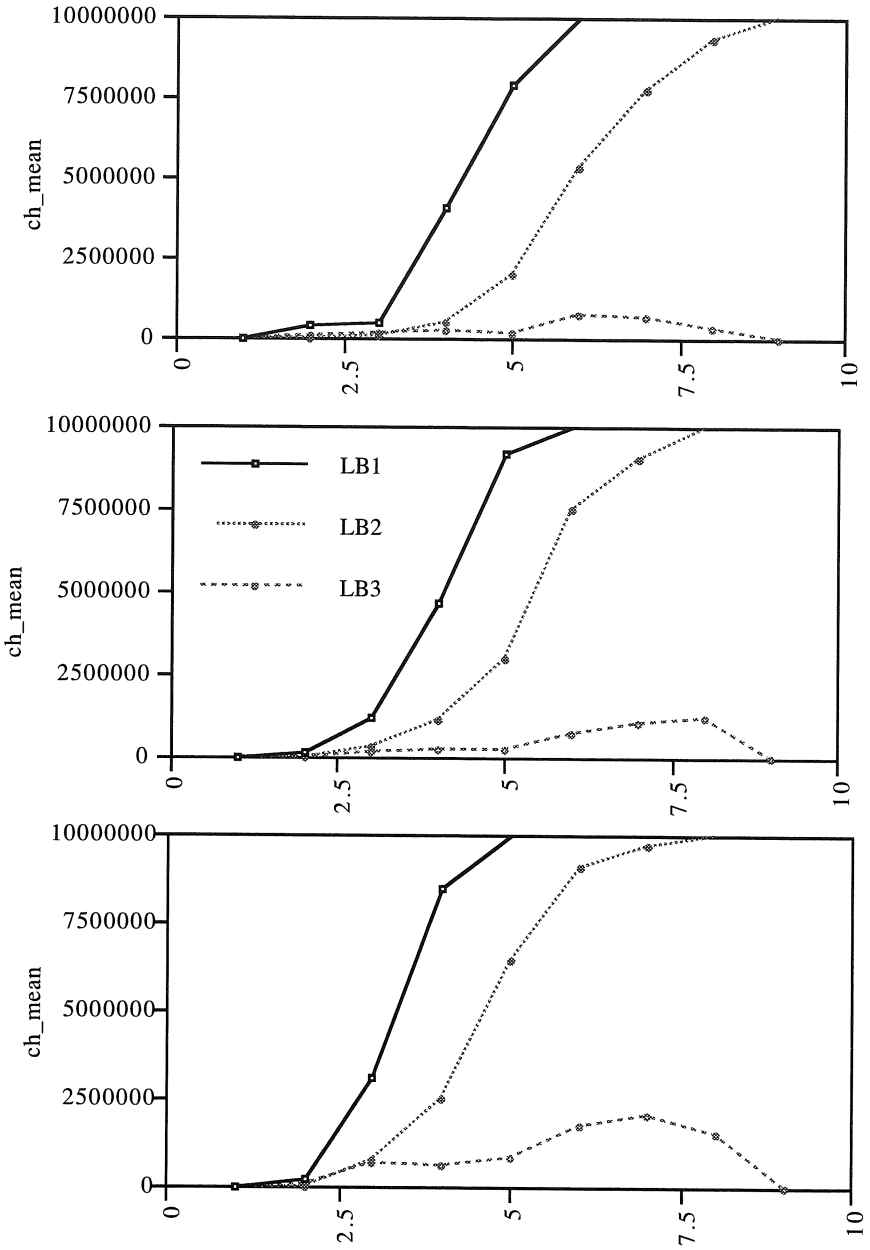


Figure A.4: Experimental results of branch and bound using three different lower bounds on the <30, 3, 0, 0, 100/435, $p_2{}^{pv}$, 5>, <30, 3, 0, 0, 120/435, $p_2{}^{pv}$, 5> and <30, 3, 0, 0, 150/435, $p_2{}^{pv}$, 5> classes of random problems (from top to bottom). Horizontal axis represents $p_2{}^{pv}$ x 9. Vertical axis stands for average number of consistency checks.

References

[Agnèse *et. al*, 95] Agnèse J., Bataille N., Bensana E., Blumstein D. and Verfaillie G. Exact and approximate methods for the daily management of an earth observation satellite. In *Proceedings of the 5th. ESA Workshop on Artificial Intelligence and Knowledge Based Systems for Space.* 1995.

[Bacchus and Grove, 95] Bacchus F. and Grove A. On the forward checking algorithm. In *proceedings of the 1st. Int. Conf. on Principles and Practice of Constraint Programming, CP-95*, 292-309, 1995.

[Bacchus and van Run, 95] Bacchus F. and van Run P. Dynamic Ordering in CSPs. In *proceedings of the 1st. Int. Conf. on Principles and Practice of Constraint Programming, CP-95*, 258-275, 1995.

[Van Beek, 91] Van Beek P. On the minimality and decomposability of constraint networks. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-92*, 447-452, 1992.

[Bellicha *et al.*, 94] Bellicha A., Capelle C., Habib M., Kökény T. and Vilarem M.C. CSP techniques using partial orders on domain values. In *ECAI-94 Workshop on Constraint Satisfaction issues raised by practical applications*, 47-56, 1994

[Bessière, 94] Bessière C. Arc-consistency and arc-consistency again. *Artificial Intelligence* , Vol.65(3), 179-190, 1994.

[Bessière *et al.*, 95] Bessière C., Freuder E.C. and Régin, J.C. Using inference to reduce arc consistency computation. In *Proceedings of the International Joint Conference of Artificial Intelligence , IJCAI-95*, 592-598, 1995.

[Bessière and Régin, 96] Bessière C. and Régin, J.C. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *proceedings of the 2th Int. Conf. on Principles and Practice of Constraint Programming, CP-96*, 61-75, 1996.

[Cheeseman *et al.*, 91] Cheeseman P., Kanefsky B. and Taylor W. M. Where the Really Hard Problems Are. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-91*, 331-337, 1991.

[Cohen *et al*, 94] Cohen D. A., Cooper M. C. and Jeavons P. G. Characterizing tractable constraints. *Artificial Intelligence* 65, 347-361, 1994.

[Cooper, 89] Cooper M. An optimal *k*-consistency algorithm. *Artificial Intelligence* , Vol.41, 89-95, 1989.

[Crawford and Baker, 94] Crawford J. and Baker A. Experimental results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-94*, 1092-1097, 1994.

[Debruyne and Bessière, 97] Debruyne R. and Bessière C. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-97*, 412-417, 1997.

[Dechter, 90] Dechter R. Enhancement schemes for constraint processing: Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence* 41(3), 273-312, 1990.

[Dechter and Meiri, 89] Dechter R. and Meiri I. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-89*, 271-277, 1989.

[Dechter and Meiri, 94] Dechter R. and Meiri I. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence* 68,211-241, 1994.

[Dechter and Pearl, 88] Dechter R. and Pearl J. Network-based heuristics for constraint satisfaction problems, *Artificial Intelligence*, 34, 1-38, 1988.

[Dent and Mercer, 94] Dent M. and Mercer R. Minimal forward checking, *Proceedings of TAI-94*, 432-438, 1994.

[Deville and Van Hentenryck, 91] Deville Y. and Van Hentenryck P. An efficient arc consistency algorithm for a class of CSP problems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-91*, 325-330, 1991.

[Dubois *et. al.*, 96] Dubois D., Fargier H. and Prade H. possibility theory in constraint satisfaction problems: handling priority, preference and uncertainty. *Applied Intelligence*, 6, 287-309, 1996.

[Fargière, 94] Fargière H. Problème de satisfaction de constraintes flexibles, application a l'ordonnancement de production. *Ph thesis, Université Paul Sabatier*, 1994.

[Frangouli *et. al.*, 95] Frangouli H., Stamatopoulos P. and Harmandas V. UTSE: Construction of optimm timetables for university courses - a CLP-based approach. In *Proceedings of the Third International Conference of the Practical Applications of Prolog*. 225-243, 1995.

[Freuder, 78] Freuder E. C. Synthesizing constraint expressions, *Communications ACM*, Vol.21, N.11, 958-966, 1978.

[Freuder, 82] Freuder E. C. A sufficient condition for backtrack-free search, *Journal of the ACM*, Vol.29, N.1, 24-32, 1982.

[Freuder, 91] Freuder E.C. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-91*, 227-233, 1991.

[Freuder and Hubbe, 1995] Eugene C. Freuder and Paul D. Hubbe. Extracting constraint satisfaction subproblems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-93*, pages 548-555, 1993.

[Freuder and Wallace, 92] Freuder E. C. and Wallace R. J. Partial constraint satisfaction, *Artificial Intelligence*, 58: 21-70, 1992.

[Freuder and Wallace, 93] Freuder E. C. and Wallace R. J. Conjunctive width heuristics for maximal constraint satisfaction. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-93*, 762-768, 1993.

[Frost and Dechter, 94] Frost D. and Dechter R. Dead-end driven learning. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-94*, 294-300, 1994.

[Frost and Dechter, 95] Frost D. and Dechter R. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-95*, 572-578, 1995.

[Frost *et al.*, 96] Frost D., Bessière C., Dechter R. and Régin J.C. Random uniform CSP generators. http://www.ics.uci.edu/~dfrost/csp/generator.html, 1996.

[Garey and Johnson, 79] Garey M.R. and Johnson D.S. *Computers and Intractability: a guide to the theory of NP-completeness.* W. H. Freeman, 1979.

[Gaschnig, 77] Gaschnig J. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-77*, 457-462, 1977.

[Gaschnig, 78] Gaschnig J. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Canadian Artificial Intelligence Conference,* 268-277, 1978.

[Geelen, 92] Geelen P.A. Dual viewpoing heuristics for binary constraint satisfaction problems. In *Proceedings of European Conference of Artificial Intelligence, ECAI-92*, 31-35, 1992.

[Gent and Walsh, 94] Gent I.P. and Walsh T. The SAT Phase Transition. In *Proceedings of European Conference of Artificial Intelligence, ECAI-94*, A. Cohn ed., 105-109, 1994.

[Gent *et al.*, 96] Gent I.P., MacIntyre E., Prosser P., Smith B. and Walsh T. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *proceedings of the 2th Int. Conf. on Principles and Practice of Constraint Programming, CP-96*, 179-193,1996.

[Ginsberg, 93] Ginsberg M. Dynamic backtracking. Journal of Artificial Intelligence Research. 1, 25-46, 1993.

[Ginsberg *et al.*, 90] Ginsberg M., Frank M., Halpin M. and Torrance M. Search lessons learned from crossword puzzles. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-93*, 762-768, 1993.

[Haralick and Elliot, 80] Haralick R. and Elliot G. Increasing Tree Search Efficiency for Constraint-Satisfaction Problems, *Artificial Intelligence*, 14(3), 263-313, 1980.

[Haralick and Shapiro] Haralick R. and Shapiro L. The consistent labeling problem: part I. *IEEE Trans. Pattern Analysis Machine Intelligence*, 1 (2), 173-184, 1979.

[Harvey, 95] Harvey W. Nonsystematic backtracking search. *Ph. D. thesis. Stanford University.* 1995.

[Harvey and Ginsberg, 95] Harvey W. and Ginsberg M. Limited Discrepancy Search In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-95*, 607-613, 1995.

[Haselbock, 1993] Haselbock A. Exploiting Interchangeabilities in constraint satisfaction problems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-93*, 282-287, 1993.

[Hummel and Zucker, 1983] Hummel R. A. and Zucker S. W. On the Foundations of Relaxation Labeling Processes, *IEEE Trans. Pattern Analysis Machine Intelligence*, 5 (3), 267-287, 1983.

[Jeavons *et al.*, 95] Jeavons P., Cohen J. and Gyssens M. A unifying framework for tractable constraints. In *proceedings of the 1st. Int. Conf. on Principles and Practice of Constraint Programming, CP-95*, 276-291, 1995.

[Keng and Yun, 89] Keng N. and Yun D. A planning/scheduling methodology for the constrained resource problem. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-89*, 998-1003, 1989.

[Kittler and Illingworth, 85] Kittler J. and Illingworth J. Relaxation labelling algorithms - a review, *Image and Vision Computing*, 3 (4), 206-216, 1985.

[Kondrak and van Beek, 97] Kondrak G. and van Beek P. A theoretical evaluation of selected bactracking algorithms. *Artificial Intelligence*, 89, 365-387, 1997.

[Konolige, 94] Konolige K. Easy to be hard: difficult problems for greedy algorithms. In *Proceedings of the International Conference on Knowledge Representation and Reasoning.* 374-378, 1994.

[Korf, 96] Korf R. Improved Limited Discrepancy Search. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-96,* 286-291, 1996.

[Kumar, 92] Kumar V. Algorithms for constraint satisfaction problems: a survey. *AI-Magazie,* Spring 1992.

[Larrosa, 97] Larrosa J. Merging constraint satisfaction subproblems to avoid redundant search. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-97,* 424-430, 1997.

[Larrosa and Meseguer, 95] Larrosa J. and Meseguer P. Optimization-based Heuristics for Maximal Constraint Satisfaction. In *proceedings of the 1st. Int. Conf. on Principles and Practice of Constraint Programming, CP-95,* 103-120, 1995.

[Larrosa and Meseguer, 96a] Larrosa J. and Meseguer P. Phase Transition in MAX-CSP. In *Proceedings of European Conference of Artificial Intelligence, ECAI-96,* 190-194, 1996.

[Larrosa and Meseguer, 96b] Larrosa J. and Meseguer P. Exploiting the use of DAC in MAX-CSP. In *proceedings of the 2th Int. Conf. on Principles and Practice of Constraint Programming, CP-96,* 308-322, 1996.

[Larrosa and Meseguer, 98a] Larrosa J. and Meseguer P. Generic CSP techniques for the job-shop problem. In *proceedings of the11th International Conference on industrial and engineering applications of artificial intelligence and expert systems, IEA-AIE-98.*

[Larrosa and Meseguer, 98b] Larrosa J. and Meseguer P. Partial Lazy Forward Checking for MAX-CSP. Submitted to the *European Conference of Artificial Intelligence, ECAI-98,* 1998.

[Larrosa *et al.*, 98] Larrosa J., Meseguer P., Schiex T. and Verfaillie G. Reversible DAC and other improvements for solving MAX-CSP. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-98.*

[Manchak and van Beek, 94] Manchak D. and van Beek P. A C library of constraint satisfaction techniques. Available by anonymous ftp from: ftp.cs.ualberta.ca:pub/ai/csp.

[Mackworth, 77] Mackworth A. Consistency in Networks of Relations, *Artificial Intelligence,* 8(1), 99-118, 1977.

[Meseguer, 97] Meseguer P. Interleaved Depth-First Search. In *Proceedings of the International Joint Conference of Artificial Iintelligence , IJCAI-97,* 1382-1387, 1997.

[Meseguer and Larrosa, 95] Meseguer P. and Larrosa J. Constraint Satisfaction as Global Optimization. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-95*, 579-584, 1995.

[Meseguer and Larrosa, 97] Meseguer P. and Larrosa J. Solving fuzzy constraint satisfaction problems. In *proceedings of the 6th. IEEE International conference on fuzzy systems.* 1233-1238, 1997.

[Miller, 90] Miller G. Five papers on WordNet. *International Journal of Lexicography*, 3(4). 1990. WordNet is available at http://www.cogsci.princeton.edu/~wn/.

[Minton *et al.*, 90] Minton S., Johnston M. D., Philips A.B. and Laird P. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-90*, 17-24, 1990.

[Minton *et al.*, 92] Minton S., Johnston M. D., Philips A.B. and Laird P. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, Vol.58, 161-205, 1992.

[Mitchell *et al.*, 92] Mitchell D., Selman B. and Levesque H. Hard and Easy Distributions of SAT Problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-92*, 459-465, 1992.

[Montanari, 74] Montanari U. Networks of constraints fundamental properties and applications to picture processing, *Information Sciences*, Vol.7, 95-132, 1974.

[Mohr and Henderson, 86] Mohr R. and Henderson T.C. Arc and path consistency revisited. *Artificial Intelligence*, Vol.28, 225-233, 1986.

[Muscettola, 94] Muscettola N. On the Utility of Bottleneck Reasoning for Scheduling. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-94*, 1105-1110, 1994.

[Pearl, 85] Pearl J. *Heuristics.* Addison-Wesley, 1985.

[Pountain, 95] Pountain D. Contraint logic programming. *Byte*, 159-160, 1995.

[Prosser, 93a] Prosser P. Hybrid algorithms for the constraint satisfaction problem, *Computational Intelligence*, 9(3), 268-299, 1993.

[Prosser, 93b] Prosser P. Domain filtering can degrade intelligent backtracking search. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-93*, 262-267, 1993.

[Prosser, 94] Prosser P. Binary constraint satisfaction problems: some are harder than others. In *Proceedings of European Conference of Artificial Intelligence, ECAI-94*, 95-99, 1994.

[Prosser, 95] Prosser P. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177. Department of Computer Science. University of Strathclyde. 1995.

[Rosenfeld *et. al*, 76] Rosenfeld A., Hummel R. A. and Zucker S. Scene labeling by relaxation operations. *IEEE Transactions on Systems Man and Cybernetics.* 6 (6), 420-433, 1976.

[Rossi *et al.*, 90] Rossi F., Petrie C. and Dhar V. On the equivalence of constraint satisfaction problems. In *Proceedings of European Conference of Artificial Intelligence, ECAI-90*, 550-556, 1990.

[Sabin and Freuder, 94] Sabin D. and Freuder E.C. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of European Conference of Artificial Intelligence, ECAI-94*, 125-129, 1994.

[Sadeh, 91] Sadeh N. Look-ahead techniques for micro-opportunistic job shop scheduling. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Sadeh *et al*, 95] Sadeh N., Sycara K., and Xiong Y. Backtracking techniques for the job shop scheduling constraint satisfaction problem, *Artificial Intelligence*, 76, 455-480, 1995.

[Sadeh and Fox, 96] Sadeh N. and Fox M. Variable and value ordering for the job shop constraint satisfaction problem, *Artificial Intelligence*, 86, 1-41, 1996.

[Schiex *et al.*, 95] Schiex T., Fargier H. and Verfaillie G. Valued constraint satisfaction problems: hard an easy problems. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-95*, 631-637, 1995.

[Schiex *et al.*, 96] Schiex T., Régin J.C., Gaspin C. and Verfaille G. Lazy arc consistency. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-96*, 216-221, 1996.

[Selman *et al.*, 92] Selman B., Levesque H. and Mitchel D. A new method for solving hard satisfability problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-92*, 440-446, 1992.

[Smith, 94] Smith B. Phase transition and the mushy region in constraint satisfaction problem. In *Proceedings of European Conference of Artificial Intelligence, ECAI-94*, 100-104, 1994.

[Smith and Cheng, 93] Smith S. and Cheng C. Slack-Based Heuristics for Constraint Satisfaction Scheduling. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-93*, 139-144, 1993.

[Torras, 89] Torras C. Relaxation and neural learning: points of convergence and divergence, *Journal of Parallel and Distributed Computing*, vol. 6, 217-244, 1989.

[Tsang, 93] Tsang E. *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[Verfaillie *et al.*, 96] Verfaillie G., Lemaître M. and Schiex T. Russian doll search for solving constraint optimization problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-96*, 181-187, 1996.

[Wallace, 94] Wallace R.J. Directed Arc Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction. *ECAI94 Workshop on Constraint Processing*, M. Meyer editor, 69-77, 1994.

[Wallace, 96a] Wallace R.J. Analysis of heuristic methods for partial constraint satisfaction problems. In *proceedings of the 2th Int. Conf. on Principles and Practice of Constraint Programming, CP-96*, 482-496, 1996.

[Wallace, 96b] Wallace M. Practical Applications of constraint programming. *Constraints. An International Journal.* Kluwer academic Pub. 1, 139-168, (1996).

[Wallace and Freuder, 93] Wallace R. J. and Freuder E. C. Conjunctive width heuristics for maximal constraint satisfaction. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-93*, 762-778, 1993.

[Walsh, 97] Walsh T. Depth-bounded Discrepancy Search. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-97*, 1388-1393, 1997.

[Yokoo, 94] Yokoo M. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-94*, 313-318, 1994.

[Yoshikawa *et al.*, 96] Yoshikawa M., Kaneko K., Yamanouchi T. and Watababe M. A constraint based high school scheduling system. *IEEE Expert*, July 1996.

[Zabih, 90] Zabih, R. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-90*, 46-51, 1990.

[Zweben and Eskey, 89] Zweben M. and Eskey M. Constraint satisfaction with delayed evaluation. In *Proceedings of the International Joint Conference of Artificial Intelligence, IJCAI-89*, 875-880, 1989.

[Zweben and Fox, 94] Zweben M. and Fox M. eds. *Intelligent Scheduling.* Morgan Kauffman, 1994.