

Implementing an NB³ Agent

Dave de Jonge

December 2015

Introduction

About This Manual

This manual explains how to implement a negotiating agent based on the NB³ algorithm. It is part of a starter package which can be downloaded from <http://www.iiia.csic.es/~davedejonge/nb3>. The starter package includes:

- This manual
- A complete example java project containing an implemented negotiation domain, and an implemented agent based on the NB³ algorithm.

The java project contains a folder named 'lib' which contains the NB³ library, as well as two other libraries that are required to run the negotiation server. Make sure that all these libraries are added to the class path when compiling the project.

We will not give an in-depth treatment of the topic of automated negotiations. We assume that you are already familiar with this topic, and that you know the meaning of terms like 'reservation value', 'aspiration level', 'bilateral negotiations', 'multilateral negotiations' and 'negotiation protocol'. Furthermore, we assume the reader is familiar with Java programming and tree search algorithms such as Branch & Bound.

About NB³

The NB³ algorithm is an algorithm that allows an agent to negotiate with other agents in domains where the number of possible deals is very large. The fact that there could be millions of possible deals means that we cannot expect the agent to know the utility values of all possible deals. Therefore,

the agents must apply a search algorithm to explore the space of possible deals.

We make the following assumptions:

- Negotiations do not need to be bilateral, they may involve up to 64 negotiating agents.
- The number of negotiating agents is fixed and known beforehand.
- There is a pre-defined set of possible world states.
- Every agent has a finite set of actions it can take to change the current world state.
- Each negotiator has an individual utility function over the set of possible world states.
- For any possible deal the agent is at least able to estimate the deal's utility value for every other agent.
- Agents can make binding agreements with each other about the actions each will take.
- Agents are selfish: each agent wants to take those actions that increase its own utility. The agents have no interest in optimizing other agents' utility functions or reaching a social optimum.
- The number of possible agreements is too large to apply exhaustive search. For example, it could be as large as 10^{100} .
- The agents negotiate about their plans of action under the Unstructured Negotiation Protocol (see Section 1.3).
- There is a fixed deadline for the negotiations which is equal for all agents and known to all agents.
- An agent that runs the NB³ algorithm makes no assumptions about the algorithms applied by the other agents it is negotiating with. In fact, the other agents may even be humans.

The set of all possible deals that the agents can make with one another is known as the *agreement space*. The idea of NB³ is that it applies a Branch & Bound algorithm to explore the space of possible deals and determine which deal is the best to propose to the other agents.

Normally, Branch & Bound is applied to determine a solution that maximizes the value of a certain utility function. In the case of negotiations however, there is not one utility function, but instead there is a separate utility function for each negotiator. Therefore, when deciding on a deal to propose the agent cannot simply propose the deal that yields highest utility to himself. Instead, it should propose a deal that also yields a reasonable

amount of utility to the other agents involved in the deal, since they will otherwise not accept it.

This means that unlike an ordinary Branch & Bound algorithm, the NB³ algorithm stores an upper- and lower- bound for the utility values of *each* agent. So if there are 5 negotiators, then each node in the search tree stores 5 upper bounds, and 5 lower bounds.

1 Preliminaries

1.1 Example Domain: The Commodity Market

Before we explain how to implement a negotiating agent, we first explain in this section the test case that we will be using throughout the rest of this tutorial.

The example domain is a domain in with 5 negotiators, let's call them Alice, Bob, Charles, David, and Eve, trading in a market. In our notation we will simply refer to them with the letters a, b, c, d, e and we will use x as a variable that can stand for any of those negotiators. The negotiators trade in 5 types of commodities, namely:

- Gold, Oil, Iron, Grain, and Wood

Each negotiator possesses a number of units of each of these commodities (his or her 'assets'), and each negotiator may be interested in different commodities. Therefore, they may negotiate in order to swap some of their commodities with others.

The initial assets of each negotiator is indicated with a vector. For example: $\gamma_a = (3, 4, 5, 7, 9)$, means that Alice has 3 units of Gold, 4 units of Oil, 5 units of Iron, 7 units of Grain, and 9 units of Wood in her possession. Similarly, the assets of Bob are denoted by γ_b , the assets of Charles by γ_c etcetera.

Each negotiator owns a factory which enables him or her to convert these commodities into some product that has a certain value expressed in Dollars (we will ignore the process of selling the product, so we will just assume that the factories directly produce Dollars). The goal for each negotiator is to earn as many Dollars as possible. However, since each negotiator has a different kind of factory, they need different amounts of each commodity to earn their Dollars. The required commodities are also denoted as vectors. For example, $\delta_a = (0, 0, 2, 2, 0)$ means that Alice needs 2 units of Iron and 2 units of Grain to earn 1 Dollar, and that she does not need any Gold, Oil or Wood at all.

Suppose for example that $\gamma_a = (0, 0, 9, 3, 0)$ and $\delta_a = (0, 0, 3, 1, 0)$, then Alice can produce 3 Dollars, because $\gamma_a = 3 \cdot \delta_a$. However, the assets may not always be an exact multiple of the requirements. For example if we have $\gamma_a = (0, 0, 9, 4, 0)$ and $\delta_a = (0, 0, 3, 1, 0)$. In that case Alice can still only produce 3 Dollars, but she will have 1 unit of Grain left after the production. Furthermore, if we have $\gamma_a = (3, 4, 5, 7, 9)$ and $\delta_a = (0, 0, 2, 2, 0)$, Alice is only able to produce 2 Dollars. Note that even though she has enough Grain to produce 3 Dollars, she cannot do so, because she doesn't have enough Iron.

Formally, if we denote the number of Dollars that a negotiator x can produce by f_x then f_x is defined as the largest positive integer, such that the following holds:

$$\forall j \in \{1, 2, 3, 4, 5\} : f_x \cdot \delta_{x,j} \leq \gamma_{x,j}$$

where $\delta_{x,j}$ and $\gamma_{x,j}$ denote the j -th entry of the vectors δ_x and γ_x respectively.

Now, the negotiators may increase their revenues by exchanging their commodities. For example, suppose we have:

$$\gamma_a = (4, 3, 5, 0, 0) \quad \delta_a = (2, 1, 4, 0, 0)$$

$$\gamma_b = (0, 8, 3, 3, 3) \quad \delta_b = (0, 3, 0, 1, 1)$$

Without negotiations, Alice can earn 1 dollar, while Bob can earn 2 Dollars. However, if Alice and Bob agree on a deal in which Alice gives 1 unit of Oil to Bob, and Bob gives 3 units of Iron in return, then the new situation will be:

$$\gamma'_a = (4, 2, 8, 0, 0) \quad \delta_a = (2, 1, 4, 0, 0)$$

$$\gamma'_b = (0, 9, 0, 3, 3) \quad \delta_b = (0, 3, 0, 1, 1)$$

Which allows Alice to produce 2 Dollars instead of 1, and allows Bob to produce 3 Dollars instead of 2. Both negotiators profit from this deal.

In the real world when you negotiate you do not always know the exact utility functions of your opponents. In fact, this is important strategic information that you normally try to hide from your opponents. Therefore, in the Commodity Market domain the agents do not exactly know the requirement vectors of their opponents. Instead, each agent receives a the requirement vector of each opponent with a little bit of random 'noise' added, so that they do have approximate knowledge of their opponents, but not precise knowledge.

1.2 Notation and Definitions

We will now introduce some definitions and notations that are necessary to understand the NB³ algorithm.

Definition 1. A negotiation domain is a tuple $\langle A, \hat{\mathcal{O}}, \hat{f}, \mathcal{E}, \epsilon_0, t_{dead} \rangle$ where A is a set of agents $A = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$, where $\hat{\mathcal{O}}$ is a tuple of sets of actions, one for each agent: $\hat{\mathcal{O}} = (\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_n)$, where \hat{f} is a tuple of utility functions, one for each agent: $\hat{f} = (f_1, f_2, \dots, f_n)$, where \mathcal{E} is a set of world states, $\epsilon_0 \in \mathcal{E}$ the initial world state, and $t_{dead} \in \mathbb{R}^+$ the deadline for the negotiations.

A negotiation domain has a fixed set of negotiating agents $A = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$. In the commodity market for example, this is the set $\{Alice, Bob, Charles, David, Eve\}$. Furthermore, there is a set of world states, which represent the possible situations. In the commodity market a world state ϵ is given as a 5x5 matrix, in which each row is the vector of assets of a negotiator:

$$\epsilon = (\gamma_a, \gamma_b, \gamma_c, \gamma_d, \gamma_e)^T.$$

The set of all possible world states \mathcal{E} is therefore the set of all 5x5 matrices with non-zero integer entries.

Each agent $\alpha_i \in A$ has a set of actions \mathcal{O}_i to its disposal, and each of these actions can be executed, causing the state of the world to change. So each action $ac \in \mathcal{O}_i$ is in fact a function $ac : \mathcal{E} \rightarrow \mathcal{E}, ac(\epsilon) = \epsilon'$, where ϵ is the current world state and ϵ' is the new world state resulting from the execution of the action.

In the commodity market, an action consists of one agent giving a number of units of a certain commodity to another agent. For example, ‘‘Alice gives 1 unit of Oil to Bob’’ is an action. Alice is the agent who executes this action ac , so it is an element of her action set \mathcal{O}_a . In this case we call Alice the ‘supplier’ of the action and Bob the ‘consumer’ of the action.¹ When this action is executed it alters the asset-vectors of both Alice and Bob and hence changes the world state ϵ to a new world state ϵ' :

$$ac(\epsilon) = ac((\gamma_a, \gamma_b, \gamma_c, \gamma_d, \gamma_e)^T) = (\gamma'_a, \gamma'_b, \gamma_c, \gamma_d, \gamma_e)^T = \epsilon'$$

The union of the sets of actions of all agents is denoted as \mathcal{O} .

$$\mathcal{O} = \bigcup_{i \in A} \mathcal{O}_i$$

¹the terms ‘supplier’ and ‘consumer’ are specific to the Commodity Market domain and may not have any meaning in other negotiation domains.

Each agent α_i has a utility function over the set of world states $f_i : \mathcal{E} \rightarrow \mathbb{R}$. In the commodity market this utility function is simply the amount of dollars the negotiator can earn. We should note that although formally the value of f_i is defined for a world state ϵ , in the case of the commodity market it only depends on the agent’s own asset vector γ_i . Furthermore, note that in the commodity market the utility functions are determined by the requirement vectors δ_i . So by specifying δ_i one indirectly specifies f_i .

Definition 2. A plan p is a set of actions: $p \subseteq \mathcal{O}$.

A plan p acts on a world state ϵ by letting all the actions $ac \in p$ act on ϵ (to keep the the discussion simple we assume that the order in which the actions are executed is irrelevant).

Definition 3. The Agreement Space is the set of all possible plans: $2^{\mathcal{O}}$.

Definition 4. The set of participating agents $pa(p)$ of a plan p is the set of all agents that have at least one action in the plan:

$$pa(p) = \{\alpha_i \in A \mid p \cap O_i \neq \emptyset\}.$$

In the commodity market a plan could be for example consist of two actions. where the first action is that Alice gives 1 unit of Oil to Bob, and the second action is that Bob gives 3 units of Gold to Alice. In this plan Alice and Bob are the participating agents.

The idea is that the agents propose plans to each other. If all participating agents of a proposed plan accept the proposal then every agent participating in it is obliged to execute his or her actions in that plan.

Since the execution of a plan brings about a new world state, we sometimes refer to the “utility value of a plan” when we actually mean the utility value of the world state resulting from executing that plan.

1.3 The Unstructured Negotiation Protocol

The most common negotiation protocol in the literature is the Alternating Offers Protocol, in which two negotiators take turns to make proposals to each other until either one accepts the last deal proposed by the other.

The problem with this protocol however, is that it only works for bilateral negotiations, and that we think it is unnecessarily restrictive to prohibit a negotiator to make a proposal while it is not his turn.

Therefore, the NB³ algorithm is developed with a simpler protocol in mind: the Unstructured Negotiation Protocol. In this protocol any agent

can make any proposal whenever it wants: they do not take turns. Therefore, when you have made a proposal, you can immediately make another proposal, even if no one has replied to your first proposal. Furthermore, you are never obliged to reply to any proposal. Apart from accepting it, rejecting it, or making a counter proposal, you can also choose to simply ignore it.

Also, unlike the Alternating Offers protocol, a deal may involve more than two agents. The deal is considered officially binding once all agents involved in the proposed plan have accepted it.

This protocol does however require the presence of a special agent known as the *Notary*. This agent records all the proposals that have been made and records who has accepted those proposals. Once a certain proposal has been accepted by all its participating agents the Notary will send a confirmation message to all participating agents to notify them that the deal is officially binding. In some cases it may happen that the negotiators accept a deal x that is inconsistent with a deal y that has been made earlier. In that case the Notary will send a message to notify that the deal y is illegal, and therefore it is not considered a binding deal.

2 How NB³ Works

As explained, the negotiating agents will propose plans to each other. So the question is: how does the agent determine which plans it should propose?

If there are only a few possible plans to propose this is easy. The agent just needs to calculate the utility value of each possible plan and sort them in order of decreasing utility. The agent can then start by proposing the plan with highest utility, and then slowly start proposing less and less profitable deals, until one of the proposals is accepted, or until one of the other negotiators makes a proposal that is good enough to accept.

However, if the agreement space contains millions of possible plans to propose the agent will not have enough time to determine the utility values of each one of them. Instead, the agent needs to apply a smart search algorithm.

The NB³ algorithm explores the agreement space by means of a heuristic tree search. Every node represents a possible plan to propose, and hence, every time it creates a new tree node, it determines the utility value of that plan for each agent participating in it.

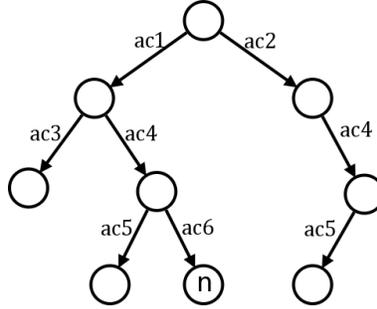


Figure 1: The search tree. The node marked n represents the partial plan consisting of the actions $ac1$, $ac4$ and $ac6$.

2.1 Expanding the Tree

The NB³ algorithm expands a search tree according to a best-first strategy. In the simplest case, each node is labeled with a certain action from the set of possible actions \mathcal{O} . Each node ν can then be identified with a plan, denoted by $path(\nu)$, that consists of all the actions that label the arcs in the path from the root to ν . In Figure 1 the node marked n represents the partial plan consisting of actions $ac1$, $ac4$ and $ac6$, so for that node we have $path(\nu) = \{ac1, ac4, ac6\}$. Moreover, we can also identify a world state ϵ_ν with this node, which is the world state that would result from letting the plan $path(\nu)$ act on the initial world state ϵ_0 .

We have here considered actions as the atomic building blocks of plans. In practice however, it is often easier to model actions as being built up from smaller constituents themselves as well. For example, in the commodity market example an action is the act of one agent giving a number of units of some commodity to another agent. Such an action can be seen as a tuple of four integers. The first integer is the ID of the supplying agent, the second integer is the ID of the commodity being given, the third integer is the number of units of that commodity that are being given, and the fourth integer is the ID of the consuming agent. In such cases it is often easier to label each node with an integer that represents one of these constituents. That is: every node with depth 1 (i.e. the children of the root) will be labeled with a supplier ID, every node with depth 2 (the grand children of the root) will be labeled with a commodity ID, nodes of depth 3 are labeled with a quantity, nodes of depth 4 are labeled with a receiver ID, and nodes of depth 5 are then again labeled with a supplier ID, etcetera.

This means that if we follow a path from the root to a leaf node, every fourth node represents an action, and only if the length of the entire path is a multiple of 4 it represents a plan.

2.2 Bounds

Branch & Bound algorithms require that each node ν stores upper- and lower bounds for the utility value of the plan corresponding to this node. In the case of negotiations, however, an agent should not only take its own utility into account but also the utility of its negotiation partners, because otherwise they would not accept any proposals.

This means that a node only represents a good proposal if it also increases the utility of the other agents sufficiently. For this reason, each node does not only compute bounds for the utility of the agent on which it is running, but also for every other agent.

The algorithm, running on agent α_1 , is thus assumed to have a model of the utility functions f_i of the other agents, and uses this model to calculate for every node ν and every agent $\alpha_i \in A$ the following bounds (we assume that the current state of the world is ϵ_0 and we use the notation ϵ_ν to indicate the world state resulting from executing the plan $path(\nu)$ corresponding to node ν . That is: $\epsilon_\nu = (path(\nu))(\epsilon_0)$).

- For each node ν and agent α_i an **upper bound**: $ub_i(\nu)$. This is the maximum utility α_i may achieve from any plan compatible with $path(\nu)$.

$$ub_i(\nu) = \max_{p \subseteq 2^{\mathcal{O}}} \{f_i(p(\epsilon_\nu)) \mid path(\nu) \subseteq p\}$$

- For each node n and agent α_i an **intermediate value**: $e_i(\nu)$. The utility agent α_i receives if exactly the plan $path(\nu)$ is executed and no other actions.

$$e_i(\nu) = f_i(\epsilon_\nu) = f_i((path(\nu))(\epsilon_0)).$$

- For each node ν and agent α_i a **lower bound**: $lb_i(\nu)$. The minimum utility that α_i may achieve from any plan compatible with the plan $path(\nu)$.

$$lb_i(\nu) = \min_{p \subseteq 2^{\mathcal{O}}} \{f_i(p(\epsilon_\nu)) \mid path(\nu) \subseteq p\}$$

The upper bounds are decreasing, and the lower bounds are increasing. That is: for any node ν and any child ν' of ν we have:

$$ub_i(\nu) \geq ub_i(\nu') \text{ and } lb_i(\nu) \leq lb_i(\nu')$$

This implies that the lower bound for agent α_i of the root node is the lowest cost agent α_i could ever achieve.

In practice, one may not always be able to calculate these bounds exactly, for two reasons. Firstly, because you may not have complete knowledge of the world state and of the other negotiators' utility functions. And secondly, because you simply do not have enough time to compute these quantities exactly in real time. Therefore, in some cases your implementation may only be able to estimate them.

The reservation value of an agent α_i is equal to the intermediate value of the root node:

$$rv_i = f_i(\epsilon_0) = e_i(\nu_0)$$

The intermediate value of a node is the utility that the agent will receive if exactly the actions in the path from this node to the root node are executed. So if $e_i(\nu) < rv_i$ the plan $path(\nu)$ is not profitable for α_i . Therefore we say a node ν is *rational* for agent α_i iff $e_i(\nu) > rv_i$.

Definition 5. A node ν is **individually rational** iff it is rational for all agents participating in $path(\nu)$.

2.3 Pruning

Since NB³ performs a best-first search, we need a heuristic h that calculates a value for each node: $h(\nu) \in \mathbb{R}^+$ to rank the nodes. We call this the *expansion heuristic*. Each time after splitting a node the algorithm picks the leaf node with the highest expansion heuristic from the tree to be split next. The value of h depends on the values of the bounds defined above. The NB³ algorithm comes with a default method to calculate the expansion heuristic of a node based on its bounds. However, this method can optionally be overridden so one can implement a better one that takes more domain specific information into account.

The upper bound is used for pruning: it defines the highest utility an agent could possibly achieve in any descendant of the node. If $ub_i(\nu) < rv_i$ for some agent α_i participating in $path(\nu)$, it means that not only this plan is unprofitable for agent α_i , but also any plan that extends $path(\nu)$ would be unprofitable for α_i , so in that case agent α_i would never agree with any plan descending from node ν and therefore this node can be pruned.

Furthermore, NB³ also applies another form of pruning. Whenever agent α_i gets committed to a plan p , all actions in \mathcal{O} that are incompatible with the actions in p become unfeasible so α_i can prune all nodes that have any of the incompatible actions in their paths to the root.

2.4 Making Proposals and Accepting Proposals

Until now we have discussed how NB³ searches through the agreement space to find potential plans that can be proposed to the other negotiators. In this section we will discuss how the algorithm decides which of those plans it should propose and which of the plans proposed by the other agents should be accepted.

2.4.1 Aspiration Levels

During the negotiations the agents regularly need to make a choice between proposing a new offer, accepting an offer, or continue searching. The NB³ algorithm bases this decision on three values: the time t passed since the start of the negotiations, the normalized utility \bar{u}_α it would receive from a possible deal and the normalized utility \bar{u}_β the opponents β receive from that deal.

Definition 6. The *normalized utility* of a plan p for agent α_i , in world state ϵ_0 , is defined as the utility divided by the maximum utility it could possibly achieve: $\bar{u}_i(p) = \frac{f_i(p(\epsilon_0)) - rv_i}{ub(\epsilon_0) - rv_i}$.

Definition 7. The *opponent utility* of a plan is the product of all the normalized utilities of the other agents participating in the plan.

$$\bar{u}_{pa}(p) = \prod_{i \in pa(p) \setminus \{\alpha\}} \bar{u}_i^\alpha.$$

with the extra restriction that $\bar{u}_{pa}(p)$ is zero if \bar{u}_i is negative for any $\alpha_i \in pa(p)$.

To make a decision, agent α compares its own normalized utility and the opponent utility with two values, denoted as $m_\alpha^\alpha(t)$ (the self-aspiration-level) and $m_\beta^\alpha(t)$ (the opponent-aspiration-level) respectively, which are fixed, time dependent functions. It is important to understand that although m_β represents an aspiration level for the utility of β , this value exists in the mind of α . It is the amount of utility that α believes to be necessary to offer to β .

Definition 8. A plan p is *more selfish* than plan p' iff $\bar{u}_i(p) > \bar{u}_{pa}(p')$. For a given time instant t we say p is *selfish enough* iff $\bar{u}_i(p) > m_\alpha^\alpha(t)$. Given a set of plans P , the plan $p \in P$ that maximizes $\bar{u}_i(p)$ is called the *most selfish* plan of P .

Definition 9. A plan p is *more altruistic* than plan p' iff $\bar{u}_{pa}(p) > \bar{u}_{pa}(p')$. For a given time instant t we say p is *altruistic enough* if $\bar{u}_{pa}(p) > m_{pa}(t)$. Given a set of plans P , the plan $p \in P$ that maximizes $\bar{u}_{pa}(p)$ is called the *most altruistic* plan of P .

Notice that ‘selfish’ and ‘altruistic’ as defined here are not necessarily each other’s opposites. If plan p yields more utility than p' , for both negotiators, p is more selfish *and* more altruistic than plan p' . $m_{\alpha}^{\alpha}(t)$ is a decreasing function, and $m_{\beta}^{\alpha}(t)$ is an increasing function.

At given moments t separated by time intervals of fixed length, α decides what to do: to propose a new plan, to accept a previously proposed plan, or to continue searching for better plans (the length of these intervals is a parameter of the algorithm). The NB³ algorithm will only accept or propose any plan that is individually rational and selfish enough. It will only propose a new plan p if there is no standing proposal p' proposed to α that is more selfish than p (because then it prefers to accept p'). And from all candidate plans it could propose, it prefers the most selfish plan that is altruistic enough. If however no plan is altruistic enough, it prefers the plan that is most altruistic. The algorithm intends to find plans for which the opponent-utility is as close as possible to the opponent-aspiration-level. The self-aspiration level imposes an extra criterion, that determines whether the plan is selfish enough to be proposed, or whether it is better to continue searching instead.

The aspiration levels have the following expressions:

$$m_{\alpha}^{\alpha}(t) = 1 - \frac{e^{-a_1 \frac{t}{t_{dead}}} - 1}{e^{-a_1} - 1} \quad (1)$$

$$m_{\beta}^{\alpha}(t) = \frac{e^{-a_2 \frac{t}{t_{dead}}} - 1}{e^{-a_2} - 1} \quad (2)$$

Their graphs are plotted in Figure 2. Notice that m_{α}^{α} decreases from 1 to 0 and m_{β}^{α} increases from 0 to 1. The higher the values of a_1 and a_2 , the faster the agent concedes. Therefore a_1 and a_2 are called the *concession degrees*. The strategy of the agent can be adapted by adjusting these two parameters. The default values for the concession degrees are $a_1 = 2$ and $a_2 = 4$.

The fact that m_{α}^{α} and m_{β}^{α} go to 1 and 0 respectively makes this strategy a very weak one for bilateral negotiations, since it can be easily countered by any opponent. The opponent β would simply not concede, but wait until m_{β}^{α} is so high that α will propose a plan that is highly favorable to

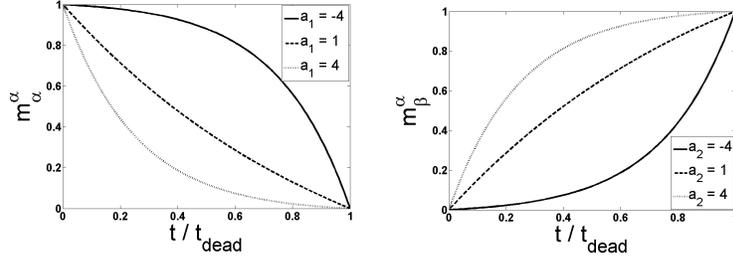


Figure 2: The graphs of m_α^α and m_β^α for several values of a_1 and a_2 .

β . However, one should keep in mind that this strategy is developed for multilateral negotiations. In the multilateral case, agent β does not have the opportunity to wait until α makes a highly altruistic offer, since β has competition from other agents. If β does not concede, α might reach deals with some of the other agents, leaving β with nothing.

Finally, note that this strategy never rejects any proposal. Instead, it simply ignores bad proposals.

2.4.2 Characterization of Strategies

We will now discuss how different values of the concession degrees affect the negotiation strategy. We define four strategies by setting the values of a_1 and a_2 either high or low.

Greedy: low a_1 , low a_2 . Only proposes very selfish plans. If it hasn't found any plans that are selfish enough, it prefers to continue searching for them than to concede.

Lazy: high a_1 , low a_2 . Proposes very selfish plans, but if it can't find any, it will propose less selfish plans, rather than to search for better solutions.

Picky: low a_1 , high a_2 . This agent is willing to propose altruistic plans, but only if they are also selfish, otherwise it prefers to continue searching. So it keeps searching until it finds a plan that is both very selfish and very altruistic.

Desperate: high a_1 , high a_2 . Concedes fast, even if this will yield low utility.

Roughly we can say that the higher the value of a_1 , the less the agent likes to search. The higher the value of a_2 , the more altruistic the plans are that the agent proposes (or is willing to accept). The Greedy strategy should

only be played if the agent has little competition. If the agent knows it has a stronger position than its opponents it can use this strategy to exploit them. The Desperate strategy on the contrary, should only be played in a highly competitive environment. If there is a lot of competition it is better to try to come to an agreement as soon as possible, before the competition takes away all the good deals.

The Lazy and Picky strategies are more moderate. The Lazy strategy should be played if good plans are scarce. In such an environment it is not likely to find many plans that are better than your current options, so it is better to give up some utility than to continue searching for a better plan. If good plans are abundant, it is better to play the Picky strategy. In that case, if your current options are not good enough, instead of giving up utility it is better to keep searching a bit more because it is likely that you will find some better plan.

3 Implementation

In this section we will explain the implementation of a negotiating agent that we have implemented for the Commodity Market example, based on the NB³ algorithm.

3.1 Commodity Market Java Classes

The CommodityMarket project contains two packages, named **commodityMarket.domain** and **commodityMarket.agent**. The first package contains all the classes that define the Commodity Market domain. They allow you to start the market and run the agent implemented for the market. Note that these classes do not assume anything about the implementation of their agents. Specifically, they are completely independent of the NB³ library. This allows anyone to implement any kind of agent they like, not necessarily based on the NB³ algorithm.

In the commodityMarket.domain package you will find a class named CommodityMarket with a main method. When running this application a random initial assignment of commodities is generated, represented by an instance of the CommodityAssets class, as well as a random utility profile (the requirement vectors), represented by the CommodityUtilityProfile class. Then, it creates a Notary agent and starts a negotiation server which allows agents to connect through a TCP/IP connection and participate in the negotiations. Note that this application can also be run as a thread, rather than as a process.

The second package is the package where we have put the implementation of our agent. The main class of the agent is `Negotiator`. The classes with a name that starts with `Cm` are derived from classes in the `NB3` library.

In order to run a negotiation session you should first run the `CommodityMarket` and then run five agents. When starting an agent it will automatically connect to the negotiation server and will start negotiating with the other agents that are connected (it assumes the server is running on localhost and on port 1234). For convenience we have added a class named `RunMarketAndFiveAgents` with a main method. You only need to run this application, and it will start a `CommodityMarket` as a thread for you, as well as five instances of our agent, each also as a thread.

When the deadline finishes you will see that a number of log files will have appeared at the path indicated in the `RunMarketAndFiveAgents` class. For each negotiating agent there will be two log files: one containing information about the nodes in its search tree (all the bounds, the normalized utility and the expansion heuristic), and one file containing information about the deals it has or has not proposed and accepted.

Furthermore, there is a log file for the server. This file contains all the messages sent and received during the negotiations (including all proposals made and accepted), the initial assets of the agents, their requirement vectors, and the final outcome of the negotiations which is displayed as a table that lists for each agent its initial utility and its final utility. If everything is okay then no agent should have a final utility that is lower than its initial utility. Furthermore, if successful deals have been made, then at least one agent participating in every such deal should have a final utility that is higher than its initial utility.

Since the domains are randomly generated, it may occur that no profitable deals are possible, or that it is simply too hard to find any profitable deals. Therefore, you may need to run a couple of negotiation sessions before you see any positive result.

3.2 Implementing the `WorldState` and `Action` classes

Before implementing the agent itself, we have to make sure that the `NB3` algorithm can ‘understand’ the `Commodity Market` domain. This means that we need to implement some classes that will form a bridge between the domain and the `NB3` algorithm.

`NB3` assumes that any domain is described by an initial world state, and that agents can execute actions to alter that state. The `NB3` library therefore contains abstract classes `NB3WorldState` and `NB3Action` that need to be

implemented according to the domain.

Step 1: implement a WorldState class, derived from NB3WorldState

Our first step is in implementing an instance of NB³ is to implement a class that will represent the world state. Note that the domain package already contains such a class, namely `CommodityAssets`. However, the NB³ algorithm requires a class that extends the `NB3WorldState` class. Therefore, we have created a new class named `CmWorldState` which is essentially just a wrapper around the `CommodityAssets` class, but extends the `NB3WorldState` class. We still need to implement a number of abstract methods, which we will do below in step 3.

```
public class CmWorldState extends Nb3WorldState{
    CommodityAssets commodityAssignment;

    CmWorldState(CommodityAssets commodityAssignment){
        super(CommodityAssets.NUMAGENTS);
        this.commodityAssignment = commodityAssignment;
    }

    public Nb3WorldState copy() {
        //TODO: implement this
    }

    public boolean isLegal(List<? extends Nb3Action> actions) {
        //TODO: implement this
    }

    public void update(ArrayList<? extends Nb3Action> actions) {
        //TODO: implement this
    }
}
```

Step 2: implement an Action class, derived from NB3Action

Next, we need to create a class that will represent an Action. Again, the domain classes already contains such a class, namely `CommodityTransaction`, but since NB³ requires a class that extends the `NB3Action` class we have created a new class named `CmAction` which is essentially a wrapper around the `CommodityTransaction` class, but extends `NB3Action`.

We now need to implement a method that returns the set of agents that are involved in this action. The NB³ library specifies a class called `NB3AgentSet` that represents a set of agents. It assumes that each agent

is identified with a number from 1 to n (n is the number of agents taking part in the negotiations). Since the Commodity Market makes the same assumption it is again easy to implement this method. We just get the ID of the supplying agent and the id of the consuming agent and put them into this set.

IMPORTANT: as we will discuss later, the NB³ algorithm is initialized with an array of Strings that maps the ID of each agent to the name of each agent. We need to make sure that this array is the same as the array that is received from the Notary at the beginning of the negotiation session, otherwise the IDs used internally by the NB³ algorithm would not match the IDs used by the Notary agent.

```
public class CmAction extends Nb3Action {

    CommodityTransaction transaction;

    public CmAction(CommodityTransaction transaction, String[] agentNames){
        this.transaction = transaction;
    }

    @Override
    protected NB3AgentSet determineParticipatingAgents() {

        NB3AgentSet pa = new NB3AgentSet();
        pa.add(transaction.getConsumer());
        pa.add(transaction.getSupplier());

        return pa;
    }
}
```

Step 3: implement the methods of the WorldState class

We can now implement the methods of our WorldState class. There are three methods:

- copy() should make a copy of the world state.
- isLegal() verifies whether it is possible to execute the given list of actions on this world state.
- update() updates the WorldState object according to the given actions.

```
public Nb3WorldState copy() {
    return new CmWorldState(this.commodityAssets.copy());
}

@Override
public boolean isLegal(List<? extends NB3Action> actions) {
```

```

//check if the bid is valid

//Create a temporary copy of the current CommodityAssets object.
CommodityAssets tempCopy = this.commodityAssets.copy();

for(NB3Action action : actions){
    CmAction cmAction = (CmAction)action;

    //Let the action act on the assets.
    tempCopy.exchange(cmAction.transaction);

}

//Now check that no agent has a negative amount of any commodity.
for(int agentID=0; agentID<commodityAssets.NUMAGENTS; agentID++){
    for(int commodity=0; commodity<commodityAssets.NUMCOMMODITIES; commodity++){
        if(tempCopy.getAssets(agentID, commodity) < 0){
            return false;
        }
    }
}

return true;
}

public void update(ArrayList<? extends Nb3Action> actions) {
    for(Nb3Action action : actions){
        commodityAssets.exchange(((CmAction)action).transaction);
    }
}
}

```

We have seen that in this case the implementation of our derived World-State and Action classes was very simple, because the Commodity Market domain already contained two classes representing world states and actions. If you want to write an NB³ agent for another domain you may need to be a bit more creative in order to fit the model of that domain with the worldstate/action model of NB³.

3.3 Implementing the NB³ Algorithm

We are now ready to start implementing the actual algorithm. For this we need to create a class, which we will call CmNB3Algorithm, which extends the class NB3Algorithm.

3.3.1 Implement getChildNodeType()

As explained above, the NB³ algorithm expands a search tree in which each node is labeled with an action, or with a smaller component of an action. In this tutorial we will make use of the fact that actions in the Commodity Market domain consist of four components:

- the supplier
- the commodity
- the consumer
- the number of units

We will assume the following:

- The children of the root node (the nodes of depth 1) will all be labeled by the name of an agent. These nodes are called *supplier nodes*.
- The children of any supplier node will be labeled with the name of a commodity, and will therefore be called *commodity nodes*.
- The children of any commodity node will again be labeled with the name of an agent, but these nodes will be called *consumer nodes*.
- The children of any consumer node will be labeled by an integer number. These nodes will be called *quantity nodes*.
- The children of any quantity node will again be supplier nodes, and thus be labeled by the name of an agent

Each node is an object of class NB3Node, and has methods getType() and getLabel(). The first will return an integer that identifies which type of node it is (e.g. a supplier node or a commodity node), and getLabel() returns the label of the node.

We first associate an integer to each type of node. The root node always has type 0, while all other nodes have types 1 to n , where n is the number of node types defined. So in this case, the supplier nodes have type 1, the commodity nodes type 2, the consumer nodes type 3, and the quantity nodes type 4. Then, in order to make sure that the NB³ algorithm indeed adds the right type of child node under the node to split, we have to implement getChildNodeType().

We should also make sure that in the constructor of CmNB3Algorithm we call the constructor of the parent class with the value 4, to tell the parent class that there are 4 types of nodes.

```
final static int ROOT = 0;
final static int SUPPLIER = 1;
```

```

final static int COMMODITY = 2;
final static int CONSUMER = 3;
final static int QUANTITY = 4;

//we have defined 4 node types (apart from the root).
final static int NUMBER_OF_NODE_TYPES = 4;

CmNB3Algorithm(Negotiator agent){
    //Make sure the parent class knows how many node types we have defined.
    super(NUMBER_OF_NODE_TYPES);

    this.theAgent = agent;
}

@Override
protected int getChildNodeType(int typeOfNodeToSplit){

    switch (typeOfNodeToSplit) {
    case ROOT:
        return SUPPLIER;
    case SUPPLIER:
        return COMMODITY;
    case COMMODITY:
        return QUANTITY;
    case QUANTITY:
        return CONSUMER;
    case CONSUMER:
        return SUPPLIER;
    default :
        throw new IllegalArgumentException("ExampleNB3Algorithm.getChildNodeType() _Error! _unk
    }
}

```

3.3.2 Implement getSplitLabels()

In every iteration of the algorithm, a node is picked from the tree to be expanded. In order to add children to this node, the algorithm needs to know which labels it must add to the children, so it will call a method called `getSplitLabels()`. We should implement this method such that it returns the list of commodities when the selected node is a supplier node, and such that it returns a list of consumer names when the selected node is a commodity node, etcetera.

```

protected List<Object> getSplitLabels(NB3Node nodeToSplit) {

    //this happens if there is no more node to split in the queue.
    if(nodeToSplit == null){

```

```

    return null;
}

List<Object> childLabels = new ArrayList<>();

CmWorldState worldState = (CmWorldState) this.getCurrentState();
int numAgents = worldState.commodityAssets.NUMAGENTS;
int numCommodities = worldState.commodityAssets.NUMCOMMODITIES;

int supplierID;
int commodityID;

int childNodeType = getChildNodeType(nodeToSplit.getType());

switch (childNodeType) {
case SUPPLIER: //add SUPPLIER nodes

    for (supplierID=0; supplierID<numAgents; supplierID++){

        //verify that we can add it.
        if( ! canBeAdded(nodeToSplit , supplierID)){
            continue;
        }

        childLabels.add(supplierID);
    }

    break;
case COMMODITY: //add COMMODITY nodes

    supplierID = (int) nodeToSplit.getLabel();

    for (commodityID=0; commodityID<numCommodities; commodityID++){

        //verify that we can add it.
        if( ! canBeAdded(nodeToSplit , commodityID)){
            continue;
        }

        //You can't supply this commodity if you don't have any units of this commodity.
        if(worldState.commodityAssets.getAssets(supplierID , commodityID) == 0){
            continue;
        }

        //if yes, then add it.
        childLabels.add(commodityID);
    }
}

```

```

}

break;
case QUANTITY: //add QUANTITY nodes

//get the current distribution of assets.
CommodityAssets assets = worldState.commodityAssets;

//get the id of the supplier given in the grand parent node
supplierID = (int)nodeToSplit.getParent().getLabel();

//get the id of the commodity given in the parent node.
commodityID = (int)nodeToSplit.getLabel();

//now find out how many units of this commodity the supplier owns.
int theSupplierOwns = assets.getAssets(supplierID, commodityID);

int maxValue = 10;
if(theSupplierOwns < maxValue){
    maxValue = theSupplierOwns;
}

//create a label for each possible quantity
for(int quantity=1; quantity<=maxValue; quantity++){

    Integer label = new Integer(quantity);

    //verify that we can add it.
    if( ! canBeAdded(nodeToSplit, label)){
        continue;
    }

    childLabels.add(label);
}

break;
case CONSUMER: //add CONSUMER nodes

supplierID = (int)nodeToSplit.getParent().getParent().getLabel();
commodityID = (int)nodeToSplit.getParent().getLabel();

for(int consumerID=0; consumerID<numAgents; consumerID++){

//the consumer cannot be the same as the supplier.
if(consumerID == supplierID){
    continue;
}
}

```

```

//It doesn't make sense to supply anything to someone who
//doesn't need that commodity.
if(theAgent.preferenceProfile.getRequirements(consumerID, commodityID) == 0){
    continue;
}

//verify that we can add it.
if( ! canBeAdded(nodeToSplit, consumerID)){
    continue;
}

childLabels.add(consumerID);
}

break;
default:
throw new IllegalArgumentException("ExampleNB3Algorithm.getSplitLabels() _Error! _unkno
}

return childLabels;
}

```

3.3.3 Implement getParticipatingAgents()

Whenever the NB³ algorithm creates a new node, it must calculate the value of the deal it represents for each agent that is involved in that deal. In order to determine which agents are involved, it calls the method `getParticipatingAgents()` which we should implement. The argument of this method is the branch of the tree that runs from the root node down to the new node to create. It must return the list of names of all agents participating in the deal.

We implement this, simply by looping over the given branch. If a node is a consumer node or a supplier node, then we know its label represents the name of an agent involved in the transaction, and hence in the deal, so we add the name to the list (unless it is already in the list).

```

public NB3AgentSet getParticipatingAgents( ArrayList<NB3Node> branch) {

    NB3AgentSet participatingAgents = new NB3AgentSet();
    for(NB3Node node : branch){

        if(node.getType() == SUPPLIER || node.getType() == CONSUMER){
            int agentID = (int)node.getLabel();
            participatingAgents.add(agentID);
        }
    }
}

```

```

    return participatingAgents;
}

```

3.3.4 Implement branch2actions()

Whenever the NB³ algorithm finds a node ν that represents a deal that can be considered to be proposed, the algorithm needs to create a NB3Proposal object so that it can be stored and when the time is right proposed. In order to do this it needs to collect the labels in the branch from the root to ν and convert them into NB3Action objects. To accomplish this the algorithm calls the method branch2actions().

```

public List<? extends NB3Action> branch2actions(List<NB3Node> branch) {

    List<CmAction> actions = new ArrayList<CmAction>();

    int supplier = -1;
    int commodity = -1;
    int quantity = -1;
    int consumer = -1;

    for(NB3Node node : branch){

        if(node.getType() == SUPPLIER){

            supplier = (int)node.getLabel();

        }else if(node.getType() == COMMODITY){

            commodity = (int)node.getLabel();

        }else if(node.getType() == QUANTITY){

            quantity = (int)node.getLabel();

        }else if(node.getType() == CONSUMER){

            consumer = (int)node.getLabel();

        }

        //Every 4th node in the branch represents a new action. Therefore, after every
        //4th node we collect the values of the previous 4 nodes and put them together
        //into an action.
        if(node.getType() == NUMBER_OF_NODE_TYPES){

```

```

if(supplier == -1 || commodity == -1 || consumer == -1 || quantity == -1){
    throw new IllegalArgumentException("branch2actions() Error! the components of the C
}

//now that we have all labels, we can create a transaction object
CommodityTransaction transaction =
new CommodityTransaction(supplier, commodity, quantity, consumer);

//wrap the transaction object into a CmAction.
CmAction action = new CmAction(transaction, getAgentNames());

//store the action in the list.
actions.add(action);

//reset these values.
supplier = -1;
commodity = -1;
quantity = -1;
consumer = -1;

}

}

return actions;
}

```

3.3.5 Implementing actions2Labels()

Whenever the NB³ algorithm receives a proposal from another agent, the proposal must be converted into a list of labels, so that a new branch can be added to the tree that represents this proposal.

```

public List<Object> actions2Labels(List<? extends NB3Action> actions) {

    List<CmAction> cmActions = (List<CmAction>)actions;

    //make sure that the actions are in the right order so that they can be added
    //to the tree in the right order.
    //This is not strictly necessary, but if not, it could mean we are adding a deal
    //to the tree that is already represented in a different branch.
    Collections.sort(cmActions);

    //create a list to put the labels in and return.
    List<Object> labels = new ArrayList<>(cmActions.size() * NUMBER_OF_NODE_TYPES);
}

```

```

for(CmAction action : cmActions){

    //Important! these labels should be added in exactly this order,
    //because this is also the order in which the tree search adds
    //labels to the tree,
    labels.add(action.transaction.getSupplier());
    labels.add(action.transaction.getCommodity());
    labels.add(action.transaction.getQuantity());
    labels.add(action.transaction.getConsumer());

}

return labels;
}

```

3.3.6 Implementing the Bounds

The most important part of the implementation of the algorithm is the calculation of the upper- and lower- bounds and the intermediate value. Every time the algorithm creates a new node ν it calls the methods `calculateIntermediateValue()`, `calculateUpperBound()` and `calculateLowerBound()` to calculate the bounds for each agent for that node. The algorithm passes a list to these methods that contains the branch of nodes from the root to the node ν (excluding the root). So the last node in this list is the new node ν , the second last node in this list is the parent of ν , the third last node is the grand parent of ν , etcetera, and the first node in the list is a child of the root node.

Let's start with the intermediate values $e_i(\nu)$. We first convert the branch into a list of `CmAction` objects and then let those actions act on a copy of the current world state. Note that this only takes the first $4 \cdot n$ nodes of the branch into account, for some integer n . Therefore, if the depth of node ν is not a multiple of 4 its intermediate value will be equal to its parent's intermediate value.

We can refine this calculation however, for Quantity nodes. If a node ν is a quantity node then ν and its parent ν' and its grandparent ν'' form a partially defined action that specifies that a certain negotiator will give away a certain amount of units of a certain commodity. The only thing we do not know yet is which other negotiator will be the consumer. However, even without knowing the consumer we can already subtract the commodity units from the supplier's assets.

```

@Override
public float calculateIntermediateValue(int agentID, List<NB3Node> branch, NB3WorldStat

```

```

//convert the branch into a list of actions
List<CmAction> actions = (List<CmAction>)this.branch2actions(branch);

//crate a copy of the current world state and let those actions modify this copy.
CmWorldState tempState = (CmWorldState)this.getCurrentState().copy();
tempState.update(actions);

//If the last node of the branch is a QUANTITY node, then we can refine
//the calculation, because we can subtract a number of units of a
//certain commodity from its assets. (however we do not know to which consumer
//this will be given)
if(branch.size() > 0){
    NB3Node lastNodeInBranch = branch.get(branch.size() - 1);
    if(lastNodeInBranch.getType() == QUANTITY){

        //get the supplier.
        int supplierID = (int)lastNodeInBranch.getParent().getParent().getLabel();

        //if the supplier is not the same as the agent for which we are calculating
        //the intermediate value, then this will not have any effect.
        if(supplierID == agentID){

            int quantity = (int)lastNodeInBranch.getLabel();
            int commodityID = (int)lastNodeInBranch.getParent().getLabel();

            int oldStockSize = tempState.commodityAssets.getAssets(supplierID, commodityID);
            int newStockSize = oldStockSize - quantity;
            tempState.commodityAssets.setAssets(supplierID, commodityID, newStockSize);
        }
    }
}

//Now use the assets of the copied world state to calculate the utility value
//of the agent.
int value = theAgent.preferenceProfile.calculateValue(agentID, tempState.commodityAssets);

return (float)value;
}

```

Now, in order to calculate the upper bound of an agent, one should realize that the absolute maximum you could ever achieve is when all the other agents give all their assets to you. However, we can refine this, because we notice that it does not make sense to propose a deal that contains an

action in which agent α_2 receives a unit of Gold from α_1 , but also contains an action in which agent α_2 supplies a unit of Gold to another agent α_3 . Such a deal does not make sense because α_1 may just as well directly give his Gold to α_3 .

This means that to calculate the upper bound for an agent α_i , we don't have to assume that α_i receives all assets from he will receive those assets for which α_i has not yet acted as a supplier in the same branch.

```
public float calculateUpperBound(int agentID, List<NB3Node> branch, NB3WorldState ws) {
    //We can calculate the upper bound of any agent by assuming that all other
    //agents will give everything they own to the given agent.

    //However, if the given agent already has supplied a certain commodity,
    //then it doesn't make sense to also consume it in the same deal.
    //Therefore, we can ignore such transactions to refine the upper bound.

    //Make a copy of the current state and get the commodities currently owned
    //by the agent.
    CommodityAssets maximalAssets = ((CmWorldState)ws).commodityAssets.copy();

    // Let all other agents give their assets to the current agent, except when
    //the current agent has already acted
    // as a supplier for a certain commodity in the branch.
    for(int otherAgentID = 0; otherAgentID<maximalAssets.NUMAGENTS; otherAgentID++){

        if(otherAgentID == agentID){
            continue;
        }

        for(int commodity=0; commodity<maximalAssets.NUMCOMMODITIES; commodity++){

            if( ! hasSupplied(agentID, commodity, branch) ){
                //check if the given agent appears as a supplier of this commodity in
                //the given branch.

                int quantity = maximalAssets.getAssets(agentID, commodity) + maximalAssets.getAsset

                maximalAssets.setAssets(agentID, commodity, quantity);
            }
        }
    }

    //calculate the utility value of the given agent when it has received all assets
    //of all other agents and return this value as the upper bound.
    int val = this.theAgent.preferenceProfile.calculateValue(agentID, maximalAssets);
}
```

```

return val;
}

```

Similarly, we can calculate the lower bound of a node ν for some negotiator, by assuming he may give everything away that he has, except for those commodities for which he has already acted as a consumer in the branch to ν .

```

public float calculateLowerBound(int agentID, List<NB3Node> branch, NB3WorldState ws) {

    if(branch.size() < NUMBER_OF_NODETYPES){
        return 0f;
    }

    //If agent A has received a certain commodity, then it doesn't make sense to
    //give it away again in the same deal.
    // Therefore, the lower bound can be calculated by removing all commodities
    //the agent has not received.

    //Make a copy of the current state and get the commodities currently owned
    //by the agent.
    CommodityAssets assets = ((CmWorldState)ws).commodityAssets.copy();
    int[] agentAssets = assets.getAssets(agentID);

    // set all its commodities to zero, except those for which the given agent has acted a
    for(int commodity=0; commodity<assets.NUMCOMMODITIES; commodity++){
        if( ! hasConsumed(agentID, commodity, branch)){
            agentAssets[commodity] = 0;
        }
    }

    return this.theAgent.preferenceProfile.calculateValue(agentID, agentAssets);
}

/**
 * Returns true if the given agent has acted as a consumer for the given
 * commodity in the given branch.
 * Returns false otherwise.
 *
 * @param agentName
 * @param currentCommodity
 * @param branch

```

```

* @return
*/
boolean hasConsumed(int agentID, int currentCommodity, List<NB3Node> branch){

    for(int i=0; i<branch.size(); i+=NUMBER_OF_NODE_TYPES){

        int consumer = -1;
        int commodity = -1;

        for(int j=0; j<NUMBER_OF_NODE_TYPES; j++){

            if(i+j >= branch.size()){
                break;
            }

            NB3Node node = branch.get(i+j);

            if(node.getType() == CONSUMER){
                consumer = (int) node.getLabel();
            }else if(node.getType() == COMMODITY){
                commodity = (int) node.getLabel();
            }

        }

        if(consumer == agentID && commodity == currentCommodity){
            return true;
        }

    }

    return false;
}

```

3.3.7 Implementing negotiate()

Now, in order to get the agent negotiating, we have to write the method that initiates the algorithm. We have called this method `negotiate()` and it is called from the `main()` method (if the agent is run as a process), or from the `run()` method (if the agent is run as a thread).

The `negotiate` method first connects to the server, by calling `negoClient.connect(this.myName)` and then waits till it receives a message from the notary that the negotiation session has started, by calling `negoClient.waitTillReady()`. This method returns an array with objects that contain information (sent to the agent by the Notary) about the negotiation domain, such as the deadline, the names and IDs of the negotiators, the initial assets and the requirement vectors (as

explained above the requirement vectors returned by the notary are not the *exact* requirement vectors that determine the agents' profits but copies with a bit of random noise added, so that they only yield approximations to the actual utility functions).

It then initializes the NB³ algorithm by calling `nb3Algorithm.initialize(myName, agentNames, nb3InitialState, deadline);` Note that the array `agentNames` must be exactly the same as the one returned by the Notary, because it maps the agents' IDs to their names. This mapping is used by the Notary as well as by the NB³ algorithm and problems could occur if they would use a different mapping.

Finally, we enter the loop that implements the negotiation process.

```
//loop until the deadline is reached, or a deal has been confirmed.
while(currentTime < this.deadline){

    //Expand the search tree.
    nb3Algorithm.expand();

    //See if we have received any message.
    if(negoClient.hasMessage()){

        //if yes, remove it from the message queue.
        Message msg = negoClient.removeMessageFromQueue();

        //convert it so that the NB3 algorithm can handle it.
        NB3Message nb3msg = nb3Algorithm.convertMessage(msg);

        //let the NB3 algorithm handle the message.
        nb3Algorithm.handleIncomingMessages(nb3msg);
    }

    //Check if enough time has passed since our last 'accept-or-propose decision'
    //and if yes make another accept-or-propose decision.
    if(currentTime - time_last_decision > deliberation_time){
        time_last_decision = currentTime;
        nb3Algorithm.acceptOrPropose();
    }

    currentTime = System.currentTimeMillis();
}
```

3.4 Implementing the Communication Methods

NB³ assumes that agents propose and accept deals by exchanging messages. These messages are represented by the NB3Message class. Of course, the

domain for which you are writing an agent generally defines its own class to represent messages. Since NB³ should not assume anything about the implementation of messages in the domain, and the domain cannot know anything about the NB³ implementation, you should implement two methods that convert the messages of the domain to objects of the NB3Message class and vice versa.

3.4.1 Implementing convertMessage()

This method takes a message received from the server and must convert it into an NB3Message object so that the NB³ algorithm can handle it.

```
protected NB3Message convertMessage(Object domainMessage){
    Message domainMsg = (Message)domainMessage;

    //Convert the particle of the message into an MsgType.
    MsgType msgType;
    if(domainMsg.getPerformative().equals("PROPOSE")){
        msgType = MsgType.PROPOSE;
    }else if(domainMsg.getPerformative().equalsIgnoreCase("ACCEPT")){
        msgType = MsgType.ACCEPT;
    }else if(domainMsg.getPerformative().equals("CONFIRM")){
        msgType = MsgType.CONFIRM;
    }else if(domainMsg.getPerformative().equals("REJECT")){
        msgType = MsgType.REJECT;
    }else if(domainMsg.getPerformative().equals("ILLEGAL")){
        msgType = MsgType.ILLEGAL;
    }else{
        throw new IllegalArgumentException("ExampleAgent.convertMessage() _Error! _unknown_perf
    }

    //Extract the Proposal object from the incoming message
    Proposal proposal = (Proposal)domainMsg.getContent();

    //Extract the list of proposed transactions from the proposal.
    List<CommodityTransaction> transActions =
    (List<CommodityTransaction>)proposal.getProposedDeal();

    //Convert the transactions into NB3 type actions.
    ArrayList<CmAction> actions = new ArrayList<>();
    for(CommodityTransaction transaction : transActions){
        actions.add(new CmAction(transaction, theAgent.agentNames));
    }

    NB3Message nb3Message = new NB3Message(domainMsg.getSender(),
    domainMsg.getReceivers(), msgType, actions, proposal.getId());
}
```

```

    return nb3Message;
}

```

The NB³ algorithm assumes that each proposal is identified with an ID of type String. This is useful because it means that when accepting a proposal the NB³ algorithm only needs to provide the id of that proposal in the ACCEPT message, rather than the proposal itself and it means that when receiving an ACCEPT message it is easier for NB³ to check that it indeed corresponds to a proposal made earlier.

In our implementation of the Unstructured Negotiation Protocol we have made similar assumptions. Each message must contain a proposal ID so that it is easier for the Notary to check which proposals have been accepting by whom.

When converting a domain message to an NB3Message we can therefore simply copy the proposalID from the domain message into the NB3Message. However, when working in a different domain you may need to write code to generate a proposal ID.

3.4.2 sendMessage()

The method sendMessage() essentially does the opposite. Whenever the NB³ algorithm decides to accept a proposal or make a new proposal it calls this method. This method should then be implemented by you so that it correctly converts the given NB3Proposal object into a message that can be sent to the server.

```

protected void sendMessage(NB3Proposal nextProposal, MsgType messageType) {

    String sender = theAgent.name;
    List<String> receivers = nextProposal.getParticipatingAgents();

    String msgParticle;
    if(messageType == MsgType.ACCEPT){
        msgParticle = "ACCEPT";
    }else if(messageType == MsgType.PROPOSE){
        msgParticle = "PROPOSE";
    }

    List<CommodityTransaction> deal = new ArrayList<>();
    for(NB3Action action : nextProposal.getActions()){
        deal.add(((CmAction)action).transaction);
    }

    Message msg = new Message(sender, receivers, msgParticle, deal);
}

```

```
    this.theAgent.sendMessage(msg);  
}
```

4 Final remarks

When implementing NB³ it is best if you make sure that each possible proposal can be represented by a branch of nodes in the tree in a unique way. Otherwise the algorithm may become very inefficient, because many proposals could be represented in the tree multiple times, by different branches. In particular this may happen when a deal is represented by a set of actions for which the order is irrelevant.

5 Exercises

Exercise 1

Start a negotiation session by running the class `RunMarketAndFiveAgents`. After 30 seconds you should see the results of the negotiations.

Now go to the `logFolderPath` specified in the source code of this class and check that indeed a folder has been created with the log files for all negotiators and the server.

Open the server log file and determine how many proposals each agent has made. For each proposal message sent, the log file should contain something like:

```
sent message:  
message ID: Bob_1  
conversation ID:  
from: Bob  
to: [Charles]  
particle: PROPOSE  
content: Proposal ID: Bob1  
Participants: [Bob, Charles]  
Proposed Deal: [agent 2 gives 7 units of 1 to agent 3, agent 3 gives 10 units  
of 0 to agent 2, agent 3 gives 10 units of 2 to agent 2]
```

Of course, the details are different for each proposal. The line *particle*:

PROPOSE indicates that this is a proposal. The log will also contain messages of type REGISTER, INFORM and START. You can ignore these, they form part of the communication protocol.

If the negotiations were successful you will also find messages of type ACCEPT and CONFIRM. How many times has a proposal been accepted? How many proposals have been confirmed?

Remember that the instance of the CommodityMarket is created randomly every time you run a new negotiation session. Therefore, in some cases the agents may not find any good proposal at all, so you may need to run several negotiation sessions before anything interesting happens. If after a couple of times the agents still don't make any proposals it may be better to continue with the other exercises to improve the agent and get back to this exercise with your improved agent.

Exercise 2

Make a copy of the commodityMarket.agent package so that you can adapt the agent and change the name of the Negotiator class to whatever you like, for example MyNegotiator.

Let us improve the calculateUpperBound() method in the CmNB3Algorithm class. In the current implementation the upper bound of an agent x is calculated by assuming that all other agent will give all their assets to that agent x . However, it does not make sense to make a deal in which agent x receives a commodity c from agent y and at the same time gives some units of commodity c to another agent z . Therefore, we have excluded those commodities for which agent x has already appeared as a supplier in the same branch.

The same reasoning can also be applied for the other agents: if some agent y already appears as a consumer of commodity c in the branch, then he will not also appear as a supplier lower in the branch. Therefore we can ignore transactions in which such an agent gives this commodity to agent x . Your task is therefore to adapt calculateUpperBound() such that it will also skip such transactions.

Exercise 3

Again, make sure that you have a copy of the commodityMarket.agent package. You can continue to work with the copy of the previous exercise, or make yet another copy. Now, in the constructor of your MyNegotiator class add the following line:

```
this.nb3Algorithm.setConcessionDegrees(2, 4);
```

With this line you can set the concession degrees discussed in Section 2.4.

Replace the values 2 and 4 by anything you like. Experiment with this: run several negotiations sessions with different values and see if you get better results with different values. You can for example run 5 different agents in each session with five different settings for these values and repeat this a number of times.

In order to obtain enough data you may want to put the code of `RunMarketAndFiveAgents` that runs a negotiation session in a loop so you can run the same instance many times with different values and compare the results.