Negotiation Based Branch & Bound and the Negotiating Salesmen Problem

Dave DE JONGE and Carles SIERRA
Artificial Intelligence Research Institute, IIIA-CSIC
Campus de la Universitat Autònoma de Barcelona
{davedejonge,sierra}@iiia.csic.es

Abstract. We introduce a new multiagent negotiation algorithm that explores the space of joint plans of action: NB^3. Each negotiator generates a search tree by considering both actions performed by itself and actions performed by others. In order to test the algorithm we present a new variant of the Traveling Salesman Problem, in which there is not one, but many salesmen. The salesmen need to negotiate with each other in order to minimize the distances they have to cover. Finally we present the results of some tests we did with a simple implementation of the algorithm for this problem.

Keywords. Negotiation, Search, Negotiating Salesmen Problem

Introduction

Negotiation algorithms have frequently been used to co-ordinate autonomous agents. Although negotiation has rightly been described as a search problem [1], previously proposed negotiation algorithms have mostly focused on the utility space. These algorithms assume that given a utility aspiration level it is always possible to find a proposal that would fit that level. This is often not possible when the domains of the issues are discrete or when there are integrity constraints among them [2]. In this paper we focus on complex problems for which the classical continuity assumptions do not apply and thus solutions have to be found directly at domain level. Also, we address a number of realistic assumptions that make the application of current negotiation algorithms unfeasible:

- The space of solutions is huge, i.e. there is no possibility to exhaustively explore the set of solutions.
- Solutions improve with co-operation. Some actions are interdependent, if agents help each other they are individually better off.
- The environment is only partially observable by each agent, e.g. actions made by others may not be observable.
- The environment changes due to actions of others.
- Decisions have to be made within a limited time frame.
- Solutions may involve a large number of agents, possibly including humans.

Many difficult problems belong to this class, e.g: school time table construction [3], route scheduling for package delivery companies [4], or the board game Diplomacy [5].

In this paper we introduce a new family of Branch and Bound algorithms, namely NB^3, that use negotiation as the key element in the exploration of the joint space of solutions for a number of autonomous agents. Section 1 briefs on Branch and Bound algorithms. Section 2 explains the concept of the algorithm. In section 3 we describe a new problem that we defined specifically for testing negotiation algorithms like NB^3, and in section 4 we present some initial experiments we have performed to test the algorithm and we give their results. Finally, in Section 5 we conclude.

1. Branch & Bound algorithms in a nutshell

Branch&Bound (BB) is a general algorithm to find optimal solutions in discrete domains. Here we outline the basics of the algorithm and introduce some notation, for an in-depth description we refer to [6,7].

The objective of a BB algorithm is to find a solution x to a problem that minimizes (or maximizes) a given function $f(x)$. The algorithm incrementally generates a tree where nodes represent sets of solutions S. Children nodes represent subsets of the father $(S_1, \dots, S_i, \dots, S_n)$ forming (ideally) a partition. A BB algorithm consists of three basic operations. The first one is to split the set of solutions represented in a node into a number of subsets that become the children of the node. This operation is called *Branching*. It is clear that $min_{x \in S}\{f(x)\} = min_{S=\cup_i S_i}\{min_{y \in S_i} f(y)\}$. The second operation is the establishment of bounds, lower and upper, for the value of $f(x)$ on the elements of S_i. This step is called *Bounding*. These bounds indicate how close we are to the optimal solution.

Finally, the key idea in any BB algorithm that looks for a minimum of $f(x)$ is that if the lower bound of a node is higher than the upper bound of another node, then the former node can be ignored as it will not contain the optimal solution. This third step is the *pruning* of the tree. This recursive procedure stops when the set S contains a single element or when the lower and upper bounds get equal, i.e. all contained solutions are equally good (or bad).

An advantage of BB is that, as it progresses, the global bounds get reduced and when their interval is reduced to a reasonable size we can stop the algorithm and pick up one element randomly from the set represented by the node.

2. NB^3 basic concept

Branch and bound has mostly been used as a centralized algorithm. Distributed versions do also exist that try and exploit concurrency in the exploration of the tree [8]. However, not much work has been done on the application of BB algorithms in search problems where the splitting is based on variables that are controlled by different agents, as in the asynchronous backtracking method used in Distributed Constraint Satisfaction [2], and where there is no single function $f(x)$ to optimize but a *set* of functions, one per agent, that are not centrally known. This paper proposes an algorithm that is run by every agent in a multiagent system and that uses negotiation between agents to split and to prune nodes. Next we give the basic idea of the algorithm (for more details we refer to [9]).

We assume a number of agents $A = \{\alpha, \beta, \ldots, \omega\}$ situated in an environment $\epsilon \in \mathcal{E}$. Each agent[1] $i \in A$ has the capacity of executing a set of feasible actions in a particular environment and given a set of known commitments[2]. This set of feasible actions is denoted by $fea(\mathcal{C}_i, \epsilon) \subset \mathcal{O}$, where $\mathcal{O}$ represents the set of all possible actions by any agent in any environment and $\mathcal{C}_i \subseteq \mathcal{O}$ is the subset of actions that agent i has already committed itself to. For convenience, inaction is considered as a possible action.

At particular time instants, agents decide autonomously what actions to perform in the environment. They are endowed with private goals and thus select those actions that might be more profitable for their goals. We are assuming environments where dependencies between actions are fundamental. In other words, certain actions performed by α will only be successful if they are accompanied by certain actions performed by β. This means that agents cannot decide what to do in isolation. They are assumed to have the capability of persuading one another, via negotiation, in order to co-ordinate their actions. We don't assume any global goal, agents only use their private goals to evaluate plans. We do assume that agents know which other agents are available and their possible actions. A set of actions, that is, a joint plan, $p = O_\alpha \cup O_\beta \cup \cdots \cup O_\omega$, where $O_i \subseteq fea(\mathcal{C}_i, \epsilon)$ for all $i \in A$, executed on an environment ϵ will end up in a new environment ϵ', denoted $p(\epsilon) = \epsilon'$. Agent α will measure how good ϵ' is to its goals with the help of a private function $f_\alpha(\epsilon)$. Instead of a single function to optimize $f(\epsilon)$, as in classical BB, in a multiagent setting we are then dealing with a set of functions $\{f_\alpha(\epsilon), f_\beta(\epsilon), \ldots, f_\omega(\epsilon)\}$ each one being locally optimized by a copy of the NB^3 algorithm.

During the process agents make commitments to perform certain actions by accepting proposals and reject actions by rejecting proposals. Agents have a partial view of the commitments made as conversations may be private.

We next explain the different components of NB^3 from the perspective of agent α.

2.1. Search tree

In a multiagent setting, each agent that runs the algorithm builds its own search tree. The root node of the NB^3 search tree consists of all the possible solutions to the problem. Each agent may choose to perform a subset of its feasible actions $fea(\mathcal{C}_i, \epsilon)$. The set of all feasible actions is denoted $Fea = \bigcup_{i \in A} fea(\mathcal{C}_i, \epsilon)$, and the set of plans[3] represented by a node n, is denoted as $plans(n)$ (which is a set of subsets of Fea). As already mentioned, this is in most practical applications an intractably large set for exhaustive exploration. The children of a node form a partition of the solutions of the father node. We label the link between a father and a son with the name of an action contained in all the solutions represented by the child. A path between the root and a leaf of the tree is then a (joint) (partial) plan (i.e. those actions labeling links in the path) that guarantees, at least, the worst solution in the leaf node. Given the path from node n back to the root node, we denote by $n.path$ the set of actions in the path from all agents in A, that is $n.path \subseteq \bigcup_{i \in A} fea(\mathcal{C}_i, \epsilon)$.

[1]We use Greek letters as the names of specific agents, while we use Latin letters to refer to any undetermined agent.

[2]A commitment is the declaration of an intention to act.

[3]To keep things simple we assume here that the order in which actions are taken is irrelevant for the outcome of the state of the world. Therefore, we see a plan as a set of actions, rather than a sequence of actions. So a set of plans is a set of a set of actions.

We also assume that α is situated in an environment that has strict time limits for a decision to be made. If the tree has been explored completely and an optimal solution has been found before the deadline then the decision of what α has to do is easy: the actions corresponding to α in the path to the optimal leaf node. Otherwise, the set of actions in the path from the root to the node with the best bounds plus the known commitments is a possibly good choice for action, even if only partial, and is what NB^3 considers as the best plan. In that respect, NB^3 is an anytime algorithm that always has the so far *best* plan of action ready.

2.2. *Splitting*

We are assuming a negotiation environment in which commitments among the participants have to be made along the search process. This is so because an agent cannot wait until it finds the optimal plan before negotiating with other agents, as then it would perhaps be too late to get any commitment from them: they might have already signed commitments for incompatible actions. Therefore, a trade-off exists between optimality and commitment availability. The more we forward explore from a potential commitment of others the better, but then the less probable it is to get it. How to solve this trade-off is key in the application of NB^3 to a particular problem.

Another key element of NB^3 is the decision on which node and according to which actions α should split. The algorithm generates a tree according to a best-first search, in which the best node to expand is determined by a heuristic h. This heuristic is a fundamental parameter of NB^3. It must rank the splits to make at each node according to the path to the node $n.path$ and both the goals and the trust[4] attitude of α.

2.2.1. *Offers*

When a node is found for which the set of actions in its path to the root is considered as good enough, the agent will (i) issue as many offers as needed to get the commitments from others that are required to execute this plan, and will (ii) withdraw any standing offers that are incompatible with the offers just made.

In particular, when the heuristic h used by α chooses to split a node according to β's actions and one of the children is selected to be proposed, NB^3 issues an offer to β to get its commitment on the action labeling the arc from that child to the father. If there is a standing offer that is incompatible with that action, NB^3 withdraws it.

While waiting for the acceptance of issued offers, NB^3 keeps on expanding the tree. h should prioritize those actions of α that might be interesting to the agents with open negotiation threads in preparation for a counter-offer.

When α receives an offer from β, h should prioritize those actions contained in the offer that are compatible with those in the path to the current best node. In this way, agents help each other in focusing the search on the space of potential deals.

2.2.2. *Withdraws*

When α receives a withdraw from agent β it prunes all nodes that require those actions of β in the proposal just withdrawn. Also, any withdraw is an indication that the probability

[4]Don't forget that commitments may not be respected.

of reaching a deal with β is lower than before and consequently the actions by β should have less priority in future selections to be made by h.

When α withdraws a previously sent offer to β the probability of reaching agreements with β decreases as β might be unhappy with the decision[5] and then the selection of actions that were potentially favorable to β and that h was going to select early in the splitting process should be delayed.

2.3. Bounding

During the search, the algorithm calculates the following values:

- A *global upper bound*: gub_α. The value of the best environment the agent could achieve from the current world state, without co-operation from any other agent.
- For each node n an *intermediate value*: $e_\alpha(n)$. A value that estimates the value α will obtain if the partial plan in the path $p = n.path$ and the known reached agreements $\mathcal{C}_\alpha$ are executed and no extra actions are done. That is: $e_\alpha(n) = f_\alpha((p \cup \mathcal{C}_\alpha)(\epsilon))$.
- For each node n a *lower bound*: $lb_\alpha(n)$. An estimate of the value of the best environment reachable from the joint plans in n.
- For each other agent an estimation for its global upper bound: $\{gub^\alpha_\beta,\ gub^\alpha_\gamma, ...\}$.
- For each other agent and each node n an estimation for its intermediate value: $\{e^\alpha_\beta(n),\ e^\alpha_\gamma(n), ...\}$
- For each other agent and each node n an estimation for its lower bound: $\{lb^\alpha_\beta(n),\ lb^\alpha_\gamma(n), ...\}$

In our notation we use the convention that estimations of quantities that belong to other agents have two indices. The superscript index refers to the agent making the estimation of the quantity, while the subscript index refers to the agent this quantity belongs to. So for example gub^α_β is defined as the estimation that α makes about β's global upper bound. Since the algorithm in this paper is entirely described from the point of view of agent α we will only encounter estimated quantities for which the superscript index is the letter α.

The global upper bound can be considered as the 'acceptance level': an agent will never accept any deal if it gives him a higher cost than the global upper bound. The world 'global' refers to the fact that this value is not assigned to any specific node in the tree, but rather to the entire tree itself; it is a property of the current world state.

The intermediate value of a node is the value that the agent would get if the actions in the path from this node to the root node are executed. So if $e_\alpha(n) > gub_\alpha$ the plan corresponding to node n is not profitable for α. Therefore we say a node n is *rational* for agent α iff $e_\alpha(n) < gub_\alpha$.

Note that for general BB algorithms in which one tries to minimize, usually each node carries its own local upper bound and the global upper bound is then defined as the minimum of all the local upper bounds. In the case of NB^3 however, we don't have such local upper bounds. The reason for this is that, if we would have local upper bounds and we look at the node with the lowest local upper bound, we cannot be sure that we can actually reach this node, because other agents might not co-operate with the plan

[5]humans may be involved in the MAS.

corresponding to this node. Therefore, we are never guaranteed to be able to reach the node with the lowest local upper bounds, which makes the definition of the gub as a minimum of local upper bounds useless.

When the agent needs to choose a plan to propose, the node with the lowest value of $e_\alpha(n)$ represents the plan with the lowest cost for α so this is the plan that α would prefer to execute. However, in general α needs the co-operation of some other agents to execute it, which means that the plan should also be profitable for them. This means that, as in any other form of negotiation, α should make a trade-off between minimizing its own costs, and the costs of its negotiation partners. For this reason, agent α needs to estimate the global upper bounds, intermediate values and lower bounds of the other agents.

2.4. Pruning

The lower bound is used for pruning: it defines the lowest cost an agent could possibly achieve in any descendant of the node. If $lb_i(n) > gub_i$ for any agent i involved in the actions in $n.path$, not only is this deal unprofitable for agent i, but also any deal that could be found by further exploring the children of node n will be unprofitable for i, so in that case agent i would never agree with any deal descending from node n and therefore this node can be pruned.

Furthermore, when an offer that α has issued is accepted by agent β it means that agent β is making a commitment to perform certain actions from the set of available actions $O \subseteq fea(\mathcal{C}_\beta, \epsilon)$. All actions that are incompatible with those in O are unfeasible, therefore we can prune all nodes that have any of the incompatible actions in their paths to the root. When an offer made by α to β is rejected the nodes that assumed the actions of β in the offer are also pruned.

3. The Negotiating Salesmen Problem

In this section we describe a new variant of the Traveling Salesman Problem (TSP), which we have defined in order to test the NB^3 algorithm. We call this problem the *Negotiating Salesmen Problem* (NSP). It resembles the multiple Traveling Salesmen Problem (mTSP) as described in [10], but with the main difference that each agent in the NSP is only interested in minimizing its individual path.

The idea is that several agents (the salesmen) need to visit a set of cities. The salesmen all start in the same city, and all other cities should be visited by at least one agent. Initially, each city is assigned to one salesman that has to visit it. However, the salesmen are allowed to exchange some of their cities amongst each other, so that the agents might be able to decrease the distance they have to cover. For example: if a city v is assigned to agent α, but α prefers to visit another city v', which is assigned to agent β, then α will propose to β to exchange v for v'. If β however also prefers to have v' over v he will not accept this deal. And if no other agent wants to accept v either, then α is obliged to travel along city v. However, we impose the restriction that not all cities are allowed to be exchanged. The cities that can be exchanged are referred to as the *interchangeable cities*, while the cities that cannot be exchanged are called the *fixed cities*.

In the following, all sets we mention are finite.

Definition 1 *An instance of the NSP is a tuple $\langle G, v_0, A, F, I, \epsilon_0, t_{dead}\rangle$, which consists of: a weighted graph G, a special vertex v_0 of the graph, a set of agents A, a set of fixed cities F, a set of interchangeable cities I, an initial distribution of cities ϵ_0 and a deadline t_{dead}. These components are further explained below.*

G is a complete, weighted, undirected graph: $G = \langle V, w\rangle$ with V the set of vertices (the *cities*) and w the weight-function that assigns a cost to each edge: $w : V \times V \to \mathbb{N}$ such that it satisfies the triangle inequality:

$$\forall a, b, c \in V : \quad w(a, c) \leq w(a, b) + w(b, c)$$

One of the vertices is marked as the *home city*: $v_0 \in V$. Each agent has to start and end its trajectory in this city. We use the symbol $\bar{V}$ to denote the set of *destinations*, that is: all cities except the home city: $\bar{V} = V \backslash \{v_0\}$. The set of destinations is partitioned into two disjoint subsets: F and I, so: $\bar{V} = F \cup I$ and $F \cap I = \emptyset$. They are referred to as the set of *fixed cities* and the set of *interchangeable cities* respectively.

The set of agents (the *salesmen*) is denoted by $A = \{\alpha, \beta, ...\}$. Each destination is initially assigned to an agent, by the function $\epsilon_0 : \bar{V} \to A$. We use the symbol $\bar{V}_i$ to denote the subset of $\bar{V}$ consisting of all cities that are assigned by ϵ_0 to agent i. $\bar{V}_i = \{v \in \bar{V} | \, \epsilon_0(v) = i\}$. $\bar{V}_i$ is referred to as agent i's set of preassigned cities. The definitions above imply that for each agent its set of preassigned cities can be further subdivided into: $\bar{V}_i = F_i \cup I_i$ where F_i is defined as $\bar{V}_i \cap F$ and I_i is defined as $\bar{V}_i \cap I$.

Finally, the instance includes a real number t_{dead} which represents the deadline for the negotiations. Agents are allowed to negotiate over the assignment of cities, until this deadline has passed.

Definition 2 *A solution of an instance of NSP is a tuple $\langle \epsilon_s, T\rangle$ in which ϵ_s is a distribution of cities: $\epsilon_s : \bar{V} \to A$ such that the restrictions of ϵ_0 and ϵ_s to F are equal: $\forall v \in F : \epsilon_0(v) = \epsilon_s(v)$. $T = (T_\alpha, T_\beta, ...)$ is a tuple of finite sequences of cities, one for each agent, such that for each agent i, T_i contains v_0 and all cities in $\bar{V}_i'$, with $\bar{V}_i' = \{v \in V | \epsilon_s(v) = i\}$.*

This means that in the solution, the cities are distributed between the agents according to ϵ_s, but the fixed cities F are still assigned to their original agents (they cannot be exchanged). So in the solution, the cities are redistributed: $\bar{V} = \bar{V}_\alpha' \cup \bar{V}_\beta' \cup ...$, but the fixed cities are not: $\bar{V}_i' \cap F_i = F_i$. A sequence T_i of the solution represents a cycle in the graph that starts and ends in v_0 and that passes all vertices in $\bar{V}_i'$. For each agent we then have a cost: $c(T_i) \in \mathbb{N}$, which is the length of the cycle. If T_i is given by $T_i = (v_0, v_1, v_2, ...v_k)$, then $c(T_i)$ is defined as:

$$c(T_i) = \sum_{j=1}^{k} w(v_{j-1}, v_j) + w(v_k, v_0) \tag{1}$$

By definition, an agent i prefers a cycle T_i^1 over a cycle T_i^2 if and only if $c(T_i^1) < c(T_i^2)$. We assume all agents are rational and therefore a solution is only feasible if for each agent the cost of the solution is less then the cost it would incur from the original distribution of cities ϵ_0. Note that, because of the fact that the graph is complete and satisfies the triangle inequality, one can always assume that the shortest path through a given subset of cities goes only through these cities, and does not pass any other city. Therefore, an agent can limit its search to paths that only visit its own cities.

4. Experiments

We have implemented an agent that applies the NB^3 algorithm to the NSP and performed some initial experiments with it, of which we will now present the results. It should be noted however, that at this moment no attempt has been made to implement the algorithm as efficient as possible. The results presented here were obtained with naïve heuristics and inefficient implementation. We therefore expect that the results can be improved a lot in the future.

We have done 3 experiments, each with 5 agents simultaneously running the algorithm. For each experiment we have created 5 instances of the NSP and for each such instance we have repeated the algorithm 3 times. So each experiment consisted of 15 runs with 5 agents simultaneously running the algorithm.

The three experiments differ in the number of interchangeable cities that were assigned to the agents. In the three experiments each agent had 5, 7 and 9 interchangeable cities assigned to it, respectively. Furthermore, in each experiment each agent had 1 fixed city, and the deadline for the negotiations was set at 60 seconds.

4.1. Creation of Problem Instances

For our experiments we created a few problem instances with obvious optimal solutions. The graphs of these problem instances are Euclidean graphs. That is: each city corresponds to a pair of 2-dimensional coordinates, and the distance between two cities is simply the 2-dimensional Euclidean distance.

Each graph is created as follows: we first create l random cities (with l the number of agents: $l = |A|$), far away from each other. Each of these cities is assigned to one of the agents as a fixed city (each agent gets exactly one fixed city). For each such fixed city we then generate m cities nearby. In this way we have created l clusters of each $m + 1$ cities. At first, all the cities of one cluster are assigned to the same agent. We refer to this assignment as the *'optimal assignment'* (it is optimal in the sense that each agent owns a set of cities which are very close to each other, so that the distance each agent has to cover is minimal). Then, for each agent we randomly choose one of its cities and interchange it with a random city from any another cluster.

After these changes each agent owns at least one city from another cluster, and on average each agent owns two cities from another cluster (because we make l swaps, and each swap involves 2 cities involved, in total $2l$ cities change owner). We refer to this new assignment as the *'initial assignment'*, because this is the assignment of the cities at the start of the algorithm. So the goal of the algorithm is to retrieve the optimal assignment from the initial assignment.

4.2. Evaluation of Results

While running the algorithm, every time when a deal between some of the agents is confirmed, we store the time of confirmation, and for each agent, the set of cities it owns

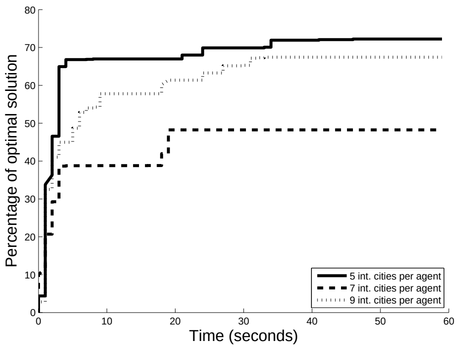

Figure 1. Results

after the deal. Also we store the initial assignment and the optimal assignment (so if k deals have been made during the execution, $l \cdot (k+2)$ sets of cities are stored).

After the algorithm has finished, for each of these sets of cities, we find the shortest path through these cities, by feeding them into the *Concorde TSP Solver* [11]. The length of the shortest path through the set of cities owned by agent i at time t is then denoted as $C_i(t)$. The length of the shortest path through the set of cities that are assigned to agent i in the optimal assignment, is denoted by C_i^*. With this notation we then define our performance measure as the following quantity:

$$Q(t) = \frac{100}{l} \sum_{i \in A} \frac{C_i(0) - C_i(t)}{C_i(0) - C_i^*} \tag{2}$$

Notice that at $t = 0$ its value is 0 by definition, and that when the agents have succeeded in retrieving the optimal assignment its value is 100 (we have included the factor of 100 in the definition so we can interpret it as a percentage).

The results of the three experiments are shown in Figure 1. What we see is that at least in two of the three experiments the algorithm has been able to find most of the deals that were necessary to retrieve the optimal assignment of cities. The fact that not all of them have been found is probably due to the fact that the algorithm uses relatively bad heuristics for estimating the length of the path through a set of cities. We expect that this can be improved.

One would expect that the results get worse as the number of cities increases, because the search space gets larger. We see however that the experiment with 7 interchangeable cities has worse results than the experiments with 5 and with 9 agents. We think that this is caused by the fact that we have used too few problem instances, so that some of the instances with 7 interchangeable cities per agent are, by coincidence, a bit more complex than the other ones. Therefore, we expect that this effect disappears when we use more instances.

5. Conclusions and Further Work

From the experiments we can conclude that the algorithm clearly works, although it is not yet optimal. However, since the results presented here were obtained using a very early version of the algorithm, there is much room for improvement. Especially we expect that we can drastically decrease the time necessary to reach agreements, by writing the code in a more efficient way.

The instances of the NSP that we used here were very simple instances, because they have obvious solutions. The cities are clearly divided in equally sized clusters, so each agent should simply collect the cities in the cluster where it has its fixed city. It would be interesting to see how the algorithm behaves with more complex problem instances.

More specifically, we should apply the algorithm to instances that involve a certain amount of competition between the agents. In the current instances, the optimal solution is highly satisfying for each agent individually, so that the problem can in fact be seen as a Distributed Constraint Optimization Problem, a kind of problem for which many good algorithms have already been invented. The point of our algorithm is however, to solve problems in which the individual wishes of the agents are not compatible. For this we first need to improve the heuristics.

Furthermore, we should compare our algorithm with other negotiation algorithms. The problem is however, that almost no algorithms have been developed so far that are capable of negotiation in large search spaces. So instead we could compare our algorithm with simplified versions of the same algorithm.

Finally, we would like to test NB^3 in other environments than NSP. For example in the game of Diplomacy (which has a huge search space) and in the problem of negotiating over time tables (in which ‘preference’ is a more abstract, non-numerical quantity).

Acknowledgements

Research supported by the Agreement Technologies CONSOLIDER project under contract CSD2007-0022 and INGENIO 2010 and by the COST Action, IC0801.

References

[1] Nick Jennings, Peyman Faratin, Alessandro Lomuscio, Simon Parsons, Carles Sierra, and Mike Wooldridge. Automated negotiation: Prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.

[2] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[3] RJ Willemen. *School timetable construction : algorithms and complexity*. T.U. Eindhoven, 2002.

[4] MJC^2. http://www.mjc2.com/transport_logistics_management.htm.

[5] Angela Fabregues and Carles Sierra. Dipgame: a challenging negotiation testbed. *Engineering Applications of Artificial Intelligence*, 2011.

[6] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[7] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[8] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, (42):1042–1066, 1994.

[9] Carles Sierra. Nb3: Negotiation-based branch and bound. Technical report, IIIA-CSIC, Bellaterra (Barcelona), Spain, 2011.

[10] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, June 2006.

[11] D. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. http://www.tsp.gatech.edu/concorde.