IJCAI 2011 Workshop

Distributed Constraint Reasoning

Barcelona, July 17, 2011

Alon Grubshtein and Pedro Meseguer, organizers.
Preface


In this edition we received 9 submissions. Each was reviewed by two members of the PC, and finally all papers were accepted for presentation. Paper contributions are on the following topics: algorithms for distributed constraint satisfaction and optimization (based on systematic and local search), algorithms based on dynamic programming, connection with graphical models, new ideas coming from social choice theory, quantification, simulation tools and platforms. All presentations are allocated 25 minutes plus 5 minutes for questions.

Alon Grubshtein & Pedro Meseguer

Workshop organizers
Program Committee

- Christian Bessiere, LIRMM-CNRS, bessiere@lirmm.fr
- Boi Faltings, Swiss Federal Institute of Technology Lausanne, boi.faltings@epfl.ch
- Alessandro Farinelli, University of Verona, alessandro.farinelli@univr.it
- Alon Grubshtein, Ben-Gurion University, alongrub@cs.bgu.ac.il
- Katsutoshi Hirayama, Kobe University, hirayama@maritime.kobe-u.ac.jp
- Robert Lass, Drexel University, urlass@cs.drexel.edu
- Amnon Meisels, Ben-Gurion University, am@cs.bgu.ac.il
- Pedro Meseguer, IIIA - CSIC, pedro@iii.csic.es
- Marius Silaghi, Florida Institute of Technology, mslaghi@fit.edu
- Evan Sultanik, Johns Hopkins University, Evan.Sultanik@jhuapl.edu
- Milind Tambe, University of Southern California, tambe@usc.edu
- William Yeoh, Singapore Management University, williamyeoh@smu.edu.sg
- Makoto Yokoo, Kyushu University, yokoo@is.kyushu-u.ac.jp
- Roie Zivan, Ben-Gurion University, zivanr@cs.bgu.ac.il
## Contents

Real-time Solving of Quantified CSPs based on MonteCarlo Game Tree Search 1  
*S. Baba, Y. Joe, A. Iwasaki, M. Yokoo*

Agile Asynchronous Backtracking for Distributed CSPs 16  
*C. Bessiere, E. H. Bouyakhf, Y. Mechqrane, M. Wahbi*

The Distributed Constraints (DisCo) Simulation Tool 30  
*A. Grubstein, N. Herschhorn, A. Netzer, G. Rapaport, G. Yafe, A. Meisels*

A Distributed Cooperative Approach for Optimizing a Network Game 43  
*A. Grubshtein, A. Meisels*

On Message-Passing, MAP Estimation in Graphical Models and DCOPs 57  
*A. Kumar, W. Yeoh, S. Zilberstein*

SOCIAL DCOP - Social Choice in Distributed Constraints Optimization 71  
*A. Netzer, A. Meisels*

Anytime Exploration for Distributed Constraint Optimization 84  
*H. Peled, R. Zivan*

Improving function filtering for computationally demanding DCOPs 99  
*M. Pujol-Gonzalez, J. Cerquides, P. Meseguer, J.A. Rodriguez-Aguilar*

DisChoco 2: A Platform for Distributed Constraint Reasoning 112  
*M. Wahbi, R. Ezzair, C. Bessiere, E. H. Bouyakhf*
Real-time Solving of Quantified CSPs based on Monte-Carlo Game Tree Search

Satomi Baba, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo
Department of ISEE, Kyushu University
Fukuoka, Japan
{s-baba@agent., yongjoon@agent., iwasaki@, yokoo@inf.kyushu-u.ac.jp

Abstract. We develop a real-time algorithm based on a Monte-Carlo game tree search for solving a quantified constraint satisfaction problem (QCSP), which is a CSP where some variables are universally quantified. A universally quantified variable represents a choice of nature or an adversary. The goal of a QCSP is to make a robust plan against an adversary. However, obtaining a complete plan off-line is intractable when the size of the problem becomes large. Thus, we need to develop a real-time algorithm that sequentially selects a promising value at each deadline. Such a problem has been considered in the field of game tree search. In a standard game tree search algorithm, developing a good static evaluation function is crucial. However, developing a good static evaluation function for a QCSP is very difficult since it must estimate the possibility that a partially assigned QCSP is solvable. Thus, we apply a Monte-Carlo game tree search technique called UCT. However, the simple application of the UCT algorithm does not work since the player and the adversary are asymmetric, i.e., finding a game sequence where the player wins is very rare. We overcome this difficulty by introducing constraint propagation techniques. We experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm. Our results show that our algorithm outperforms the state-of-the-art algorithm in large-scale problems.

1 Introduction

A constraint satisfaction problem (CSP) [10] is the problem of finding an assignment of values to variables that satisfies all constraints. Each variable takes a value from a discrete finite domain. A variety of AI problems can be formalized as CSPs. Consequently, CSP research has a long and distinguished history in AI literature. A quantified constraint satisfaction problem (QCSP) [5] is an extension of a CSP in which some variables are universally quantified. A universally quantified variable can be considered the choice of nature or an adversary. The goal of a QCSP is to make a robust plan against an adversary. A QCSP can formalize various application problems including planning under uncertainty and playing a game against an adversary.

While solving a CSP is generally NP-complete, solving a QCSP is generally PSPACE-complete. Thus, as the number of variables increases, obtaining
a complete plan off-line becomes intractable quickly when the size of the problem becomes large. In an off-line planning, if a complete solution is not found before the agent actually plays against the adversary, it is a complete failure. However, if the adversary is not omniscient, the agent does not necessarily need a complete plan to defeat the adversary. The agent should make a reasonable choice, even if it is not guaranteed to be succeed, using the available time until each deadline. In this paper, we develop a real-time algorithm that sequentially selects a promising value for each variable at each deadline.

Most existing algorithms for solving a QCSP are off-line algorithms [3, 8]. One notable exception is [12]. In [12], a real-time algorithm for solving a QCSP is presented. This algorithm applies a standard game tree search technique to QCSP; it is a combination of a lookahead method based on an alpha-beta pruning and a static evaluation function. In [12], several alternative strategies are evaluated. A strategy called Intelligent Alpha Beta (IAB) is shown to be most effective. In IAB, child nodes in a search tree are ordered from best to worst, and an alpha-beta search is executed. In this algorithm, the evaluation value is calculated based on a static evaluation function for a leaf node in a partially expanded game tree (which is not a terminal node of the fully expanded game tree).

In a standard game tree search algorithm including alpha-beta, developing a good static evaluation function is crucial. However, it is very unlikely that we can develop a good static evaluation function for a QCSP because it must estimate the possibility that a partially assigned QCSP is solvable. This task is difficult even for a standard CSP. The static evaluation function in [12] (called Dynamic Geelen’s Promise) uses the product of the sizes of future existential domains. This seems reasonable for a standard CSP, but it is not clear whether this function is really appropriate to a QCSP.

In this paper, we apply a Monte-Carlo game tree search technique that does not require a static evaluation function. A Monte-Carlo method, which is an algorithm based on repeated random sampling, evaluates the node by the results of many playouts in which we play a game randomly until it is finished. Thus, the evaluation values are stochastic. In Computer game Go, a variation of the Monte-Carlo method called the UCT (UCB applied to Trees) algorithm [9] has been very successful. One merit of UCT is that it can balance exploration and exploitation when selecting a node to start a playout. In this paper, we also use a UCT-based algorithm.

However, the player and the adversary are extremely asymmetric in a QCSP if we choose parameters, such as constraint tightness, similar to a standard CSP. A prevailing assumption in a CSP literature is that satisfying all constraints is difficult. For example, in the eight-queens problem (which is considered as a very easy CSP instance), if we place eight queens on the chess board at random, the chance that these queens do not threaten with each other is very small. Thus, if we simply apply UCT, finding a game sequence where the player wins is very rare. As a result, the UCT’s decision is about the same as a random guess. To overcome this difficulty, we introduce constraint propagation techniques based on
a concept called arc-consistency to allow the algorithm to concentrate on the part of the game tree where the player has some chance to win. We experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm (IAB). Our results show that our algorithm outperforms IAB for large-scale problems.

The rest of this paper is organized as follows. In Section 2, we show the formalization of a QCSP, real-time online solving of a QCSP, and the UCT algorithm as a related research. In Section 3, we present real-time algorithms for solving a QCSP. Then, in Section 4, we show the experimental results. Finally, in Section 5, we conclude this paper.

2 Related Research

2.1 Quantified CSP

A constraint satisfaction problem (CSP) is a problem of finding an assignment of values to variables that satisfies constraints. A CSP is described with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) constraints \( C_1, C_2, \ldots, C_m \). Each variable \( x_i \) takes a value from a discrete finite domain \( D_i \).

A QCSP [5] is a generalization of a CSP in which variables are existentially (\( \exists \)) or universally (\( \forall \)) quantified. Solving a QCSP is PSPACE-complete. Each quantifier is defined by the sequence of quantified variables. Sequence \( Q \) consists of \( n \) pairs, where each pair consists of quantifier \( Q_i \) and variable \( x_i \), as represented in (1):

\[
Q_1 x_1 \cdots Q_n x_n.
\]

(1)

Note that the sequence order matters, e.g., the meanings of \( \forall x \exists y \) loves \((x, y)\) and \( \exists y \forall x \) loves \((x, y)\) are quite different. \( \forall x \exists y \) loves \((x, y)\) means any \( x \) loves some \( y \), where \( y \) can be different for each \( x \). On the other hand, \( \exists y \forall x \) loves \((x, y)\) means particular person \( y \) is loved by everybody.

A QCSP has a form \( QC \) as in (2), where \( C \) is a conjunction of constraints and \( Q \) is a sequence of quantified variables:

\[
\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \neq x_3) \land (x_1 < x_4) \land (x_2 \neq x_3).
\]

(2)

The semantics of a QCSP \( QC \) can be defined recursively as follows:

- If \( C \) is empty then the problem is true. If \( Q \) is of the form \( \exists x_1 Q_2 x_2 \cdots Q_n x_n \), then \( QC \) is true iff there exists some value \( a \in D_1 \) such that \( Q_2 x_2 \cdots Q_n x_n C[(x_1, a)] \) is true. If \( Q \) is of the form \( \forall x_1 Q_2 x_2 \cdots Q_n x_n \), then \( QC \) is true iff for each value \( a \in D_1, Q_2 x_2 \cdots Q_n x_n C[(x_1, a)] \) is true. Here, \( C[(x_1, a)] \) is a constraint \( C \) where \( x_1 \) is instantiated to value \( a \).

2.2 Real-Time Online Solving of QCSP

In the real-time online solving of QCSP [12], a QCSP is treated as a two-players game, in which the existential player assigns values to existentially quantified
variables and the universal player assigns values to universally quantified variables. Each player must decide the value of each variable within a time limit. For the existential player, the goal is to reach a solution, but the universal player is trying to prevent a solution from being reached. Real-Time online solving of QCSP for the existential player is defined as follows:

- Given QCSP $QC$, increasing sequence of time points $t_1, t_2, \cdots, t_n$, and sequence of values $v_1, v_3, v_5, \cdots, v_{n-1}$ such that each value $v_j$ is in $D_j$ and is revealed at time $t_j$, generate at each time $t_k$ for $k = 2, 4, 6, \cdots, n$ a value $v_k \in D_k$ such that the tuple $(v_1, v_2, \cdots, v_n)$ is a solution for $QC$.

Here, for simplicity, the sequence of quantifiers of QCSP $QC$ is assumed to be a strictly alternating sequence of quantifiers, starting with $\forall$ and ending with $\exists$.

2.3 UCT

UCT (Upper Confidence bound applied to Trees) [9] is a Monte-Carlo method combined with tree search. A Monte-Carlo method is a stochastic technique using random numbers, that evaluates nodes with the mean score given by the result of many playouts in which we play a game randomly until it is finished. It is effective when a lookahead search is difficult, e.g., the size of the game tree is huge, and/or designing a good static evaluation function is hard. For Computer Go, UCT-based algorithms are very effective. CrazyStone [6] is one of the first computer Go programs that utilizes UCT. UCT is also successful in General Game Playing competitions, where an agent should accept declarative descriptions of an arbitrary game at runtime and be able to play the game effectively without human intervention [7].

UCT can be considered an application of Upper Confidence Bound (UCB) [1] to tree search, which is a technique applied to the multi-armed bandit problem. The goal of the problem is to select a sequence of arms that maximizes the sum of rewards. The reward of each arm is given by a fixed probability distribution which is initially unknown. In this problem, achieving a good balance between exploration and exploitation, i.e., whether to use a trial for gathering new information, or collecting of rewards, is important. When UCB is applied for selecting an arm, it selects an arm whose UCB value is highest. A UCB value is defined in (3):

$$\bar{X}_j + \sqrt{\frac{2 \log t}{t_j}}.$$  \hspace{1cm} (3)

$\bar{X}_j$ is the mean of rewards given by the plays of arm $j$, $t_j$ is the number of plays of arm $j$, and $t$ is the overall number of plays. The first term favors the arm with the best empirical reward while the second term prefers an arm with fewer plays.

In UCT, a partially expanded game tree (which is a subtree of a fully expanded game tree) is stored. Initially, this subtree has only one root node. UCT
continue to select a node by the UCB value in this subtree, from the root node until it reaches one leaf node of the subtree (which is not a terminal node of the fully expanded game tree). A leaf node is expanded if the number of visits of the node reaches a threshold value. When UCT arrives at a leaf node, a random playout starts from the leaf node to a terminal node, where the result of the game is determined. The result of the playout is updated iteratively from the leaf node to the root node as an evaluation value.

Figure 1 represents the subtree used in UCT. Since UCT gives only two values that represents winning or losing as a reward, an evaluation value represents the winning probability. In Figure 1, the number of visits for the left node is largest since the winning probability of the left node is highest. So the left node is expanded first. On the other hand, the center and right nodes are also visited (although less frequently than the left node) due to the second term in (3).

Algorithm 1 illustrates the UCT algorithm. UCT searches a tree repeatedly until a time limit. One search trial includes three processes. The first process (i) in Algorithm 1 is a function that selects a node by the UCB value. The second process (ii) represents one playout, which returns the result of the playout. The last process (iii) is a function that updates the evaluation values of nodes. This function updates the evaluation values from a leaf node to the root node by the result of the playout.
Algorithm 1 UCT algorithm

```plaintext
while not timeout do
    search(rootNode);
end while

function search(rootNode)
    node := rootNode;
    while node is not leaf do
        node := selectByUCB(node); – (i)
    end while
    value := playout(node); – (ii)
    updateValue(node, value); – (iii)
end function
```

3 Real-Time Algorithm based on Monte-Carlo for Solving QCSPs

In this section, we present a real-time algorithm for solving a QCSP, which is based on UCT. First, we show the basic real-time algorithm based on UCT (Section 3.1). However, since this basic algorithm fails to obtain useful information from playouts in a reasonable amount of time, we modified it by introducing constraint propagation techniques (Section 3.2).

3.1 Basic Monte-Carlo Algorithm for QCSPs

A UCT-based algorithm stores a subtree of a game tree. In a QCSP, since the values of the variables must be determined in a sequence of quantifiers, a QCSP’s game tree can be represented as follows:

- A node in a game tree represents a partially assigned QCSP. More specifically, the root node represents a state where no variable’s value has been assigned yet. A node whose depth is \( i \) represents a state where the values have been assigned from the first variable to \( i \)-th variable in the sequence.
- Each node has a variable that should be assigned next. The number of links from a node equals the domain size of the variable of the node, i.e., each link represents the choice of a value of the variable. A child node represents a state where the value of the parent node’s variable is assigned according to the link.

Our algorithm continues to select a child node (which corresponds to selecting the value of a variable by the UCB value), from the root until it reaches one leaf node of the subtree. In our algorithm, we set the threshold value of a node expansion to one, i.e., a leaf node is expanded on the first visit. When a leaf node is expanded, its children are generated and they become new leaf nodes. Then, the algorithm selects one new leaf node randomly, and starts a playout from the selected node.
In this algorithm, for each playout, we assign values randomly to all variables based on the sequence of quantifiers. The existential player wins if all constraints are satisfied, otherwise the universal player, i.e., the adversary, wins. If the existential player wins, the evaluation values of the nodes are updated by 1, otherwise by 0. The update procedure from the universal player’s point of view is symmetric. More specifically, the evaluation values of the nodes are updated by $-1$ if the existential player wins.

We introduce a Pure Value Rule [8] for basic pruning that is defined as follows:

- It prunes the search space using a concept called **pure value**, which is defined as follows:
  - Value $a \in D_i$ of QCSP QC is pure iff $\forall x_j, x_j \in Q$, where $x_j \neq x_i$ and $\forall b \in D_j$, the assignments $(x_i, a)$ and $(x_j, b)$ are compatible.

An existential variable with a **pure value** can be set to that value, while a **pure value** is removed from the domain of a universal variable.

In our algorithm, a Pure Value Rule is applied in a process selectByUCB() in Algorithm 1 when a node is expanded. The algorithm creates child nodes for values except for values removed by the Pure Value Rule, which reduces the number of child nodes since it is applied before a node is expanded. However, the Pure Value Rule is not powerful enough to reduce the search space. Since the player and the adversary are extremely asymmetric in a QCSP, the probability that a set of random value assignments satisfies all constraints is very small. Thus, finding a game sequence where the player wins is very rare, and evaluation values are updated only by 0 in most cases. As a result, the decision of UCT is about the same as a random guess. To overcome this difficulty, we introduce more powerful constraint propagation techniques so that the algorithm can concentrate on the part of the game tree where the player has some chance to win.

### 3.2 Improvement of Basic Algorithm with Constraint Propagation

We modified a basic real-time algorithm for solving a QCSP by introducing constraint propagation techniques, i.e., we remove values from the domains, that cannot be a part of the final solution. This corresponds to pruning nodes where the universal player wins in the future. By pruning such nodes, the algorithm can concentrate on the part of the game tree where the player has some chance to win. We present two different approaches on how to update evaluation values of nodes based on the result of constraint propagation: one is called **shallow** update, the other is called **deep** update.

**Incorporating More Powerful Constraint Propagation Techniques**

We introduce a constraint propagation technique based on a concept called **arc-consistency**. A CSP is called **arc-consistent**, if for any pair of variables $x_1$ and $x_2$, for any value $a$ in $D_1$, there exists at least one value $b$ in $D_2$ that satisfies the
Algorithm 2 Process of node expansion

if node is a leaf node then
    pureValueRule(node);
    for each value \( v \) in domain of variable that the node has do
        child := new node(\( v \));
    end for
    for each child do
        constraintPropagate(child);
    end for
end if

constraint between \( x_1 \) and \( x_2 \). An arc-consistency algorithm removes the values if they do not satisfy the above condition. Our algorithm achieves Strongly Quantified Generalized Arc Consistency (SQGAC) [11] by constraint propagation. For QCSP, constraint \( C \) is SQGAC iff for each variable \( x \) included in \( C \), for each value \( a \in D_x \), for all universal variables \( x_i, x_j, \ldots \), which appear after \( x \) in sequence \( Q \), and for each partial assignment \( \{(x, a), (x_i, b), (x_j, c), \ldots \} \), there exists a strategy to assign other variables in \( C \) so that constraint \( c \) is satisfied. When all constraints are binary, SQGAC is reduced to Quantified Arc Consistency (QAC) [4].

In our algorithm, when a node is expanded, or a value of a variable is determined within a random playout, constraint propagation is executed to achieve SQGAC. We present two alternative methods (deep/shallow update) that are different in how far the update of evaluation values continues.

Details of Algorithm The modified algorithms apply constraint propagation for all child nodes created as represented in Algorithm 2. The algorithm performs SQGAC. If the domain of an existentially quantified variable becomes empty or any value is removed from the domain of a universally quantified variable by achieving SQGAC, the assignment eventually leads to a constraint violation. In the shallow update method, when the algorithm finds such a constraint violation, it updates the evaluation value of the node that leads to a constraint violation to \(-\infty\).

In our deep update method, as well as the above procedure for the shallow update, we incorporate the following additional procedure to update the evaluation values of ancestor nodes.

- Assume for node \( i \), which represents an existentially quantified variable, the evaluation values of all \( i \)'s child nodes become \(-\infty\) (as a result of several shallow/deep updates). Then node \( i \) will eventually lead to a constraint violation. Thus, the algorithm updates \( i \)'s evaluation value to \(-\infty\).
- Assume for node \( i \), which represents a universally quantified variable, the evaluation value of one of \( i \)'s child nodes is \(-\infty\) (as a result of another shallow/deep update). Then node \( i \) will eventually lead to a constraint violation (assuming the adversary takes the child node whose evaluation value is \(-\infty\)). Thus, the algorithm updates \( i \)'s evaluation value to \(-\infty\).
Fig. 2. Change of winning probability with constraint propagation

Note that value $-\infty$ is used as a sign that the node causes a constraint violation. When updating the evaluation value of its parent node, it is treated as 0.

While constraint propagation is effective for pruning the search space and for concentrating playouts on the part of the game tree where the player has some chance to win, we need to redefine the way of calculating the winning probability of a node based on the result of the playouts. More specifically, when a value is removed by constraint propagation, the estimated winning probability obtained by playouts is skewed, as illustrated in Figure 2.

Here, assume 100 terminal nodes exist from the leaf node in total. Each terminal node represents a state where the values of all variables are assigned. Within 100 terminal nodes, since 40 are descendants of the removed node, they are not considered in the playouts. Assume the estimated winning probability by random playouts for this leaf node is 50%. This means that within the 60 unremoved nodes, the player can win around 30 terminal nodes. On the other hand, the correct winning probability should be $\frac{30}{40+60} = 30\%$.

To overcome this problem, we need to incorporate the information of the removed nodes, (which result in constraint violations, i.e., player’s loss) into the evaluation values and UCB values. We redefine a UCB value as follows:

$$\bar{X}_j \times \left(1 - \frac{l}{L}\right) + \sqrt{\frac{2 \log t}{t_j}}$$

(4)

$l$ is the number of terminal nodes pruned, and $L$ is the total number of terminal nodes. Thus, $l/L$ is the rate of the pruned terminal nodes (thus unexplored). Therefore, $\bar{X}_j \times (1-l/L)$ is the adjusted winning probability including the pruned terminal nodes. This probability should be close to the real winning probability.
Fig. 3. Effect of constraint propagation: QCSP against random adversary (n = 20, d = 8)

When the universal player wins in all playouts from a node, the node’s UCB value is determined only by the second term since the first term is 0.

4 Experiments

In this section, we experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm, when they play against a deliberative and random adversary. We can consider a random adversary represents a choice of nature or an irrational agent. Ideally, a real-time algorithm should perform well against both rational/irrational adversaries.

We created problem instances with a strictly alternating sequence of $\exists$ and $\forall$ quantifiers as [12]. A random binary QCSP instance is generated based on five parameters; $\langle n, d, p, p_{\exists\exists}, p_{\forall\exists} \rangle$, where $n$ is the number of variables, $d$ represents the domain size, which is the same for all variables, and $p$ represents the number of binary constraints as a fraction of all possible pairs of variables. $p_{\exists\exists}$ represents the number of constraints in the form of $\exists x_i \exists x_j, c_{ij}$ as a fraction of all possible tuples. $p_{\forall\exists}$ is a similar quantity for $\forall x_i \exists x_j, c_{ij}$ constraints, described below. The other two types of constraints are not generated since they can be removed by preprocessing.

When many constraints exist in the form of $\forall x_i \exists x_j, c_{ij}$, most problem instances are insolvable. To generate enough solvable instances, constraints in the form of $\forall x_i \exists x_j, c_{ij}$ are restricted in the following way, as described in [8].

We generate a random total bijection from one domain to the other. All tuples that are not in this bijection are excluded in the constraint. From this total bijection, choose $p_{\forall\exists}$ fraction of tuples as constraints.

In Figures 3–5, we chose the following parameters: $n = 20, d = 8, p = 0.20, p_{\forall\exists} = 0.5$. Then we varied $p_{\exists\exists}$ from 0.55 to 0.85. For each value of $p_{\exists\exists}$, we
generated 100 instances. The time limit in which each player determines a value is 1000ms, i.e., if a player uses a UCT-based algorithm, it tries to perform as many playouts as possible until the time limit. Also, if a player uses a standard game tree search, it tries to lookahead the search tree as deep as possible until the time limit. All experiments were run on an Intel Xeon 2.53GHz processor with 24GB RAM. For this parameter setting, we can check whether a problem instance has a winning strategy or not by using an off-line algorithm. When $p_{33} = 0.55$, almost all problem instances have winning strategies. When $p_{33} = 0.60$, approximately 95% of problem instances have winning strategies. Also, when $p_{33} = 0.65, 0.70,$ and 0.75, the ratios of problem instances with winning strategies are about 80%, 60%, and 20%, respectively.

Figures 3 and 4 show the ratio of problem instances that the existential player wins. Figure 3 illustrates the effect of constraint propagation. Our UCT-based algorithm without constraint propagation (MC (NoProp)) performed very badly; it is slightly better than a random player (Random). On the other hand, the performance improves significantly by incorporating constraint propagation with a shallow update method (MC (shallow)). Constraint propagation is clearly very effective in our real-time algorithm for solving QCSPs.

Figure 4 compares our UCT-based algorithms and the state-of-the-art lookahead algorithm. MC (deep) is the result of our UCT-based algorithm with a deep update method, and IAB is the result of the lookahead algorithm with an Intelligent Alpha Beta (IAB) strategy [12]. The evaluation results reported in [12] indicate that IAB with a static evaluation function called Dynamic Geelen’s Promise (DGP) [13] which uses the product of the sizes of future existential domains, performs best. Thus, we also used DGP for the static evaluation function of IAB. In Figure 4, the results of each algorithm are almost equivalent. Actually, these differences are not significant\(^1\).

\(^1\) In this section, we apply paired t-tests and the significant level is 5%.
Figure 5 shows the experimental results where the universal player applies the lookahead algorithm with Alpha Beta. As shown in Figure 5, the performance of MC (shallow) is worse than IAB and MC (deep), and these differences are significant. This is because MC (shallow) tends to explore more search space that are not reachable when the adversary is deliberative. On the other hand, MC (deep) and IAB are almost equivalent, and this difference is not significant.

In Figures 6 and 7, we show the result for larger problem instances of the following parameters: \( n = 50, d = 16, p = 0.20, p_{\exists } = 0.5 \). We varied \( p_{\exists } \) from 0.25 to 0.50 when the universal player applies a random adversary. Then we varied \( p_{\exists } \) from 0.25 to 0.45 when the universal player applies a lookahead algorithm with Alpha Beta. The time limit in which each player determines a value is 3000ms. For each value of \( p_{\exists } \), we generated 100 instances.

Figure 6 shows the experimental result against a random adversary. In this experiment, MC (shallow) performs much better than IAB and MC (deep), and these differences are significant. In a larger problem instance (with a longer time limit), the subtree stored by our Monte-Carlo algorithm becomes large. Then, performing a deep update requires certain overhead. As a result, MC (shallow) can run more random playouts than MC (deep). MC (deep) tries to avoid a value assignment that can lead to its loss by applying an additional procedure to update the evaluation values, assuming the adversary is rational. This additional effort does not pay when the adversary is a random player. IAB performs slightly better than MC (deep) but this difference is not significant.

Figure 7 shows the experimental result against Alpha Beta. In this experiment, our UCT-based algorithms performs much better than IAB, and these differences are significant. When the search tree becomes too large, a lookahead based algorithm cannot obtain enough information to make a reasonable decision. On the other hand, our UCT-based algorithm still manages to obtain some useful information within a limited amount of time. MC (shallow) and MC
Fig. 6. Large-scale QCSP against random adversary ($n = 50$, $d = 16$)

(Deep) are almost equivalent, and this difference is not significant. In this case, the additional effort used in MC (deep) is just enough to compensate its overhead since the adversary is rational. More detailed analysis is needed to clarify the merit/demerit of deep/shallow update procedures.

5 Conclusions

In this paper, we presented a real-time algorithm for solving a QCSP. We applied a Monte-Carlo game tree search technique called UCT, which has been very successful in games like Go, and does not require a static evaluation function. We found that a simple application of UCT does not work for a QCSP because the player and the adversary are extremely asymmetric and finding a game sequence where the player wins is very rare. As a result, the UCT’s decision is about the same as a random guess. Thus, we introduced constraint propagation techniques so that the algorithm can concentrate on the part of the game tree where the player has some chance to win, and obtain a better estimate of the winning probability. Experimental results showed that our UCT-based algorithm with constraint propagation greatly outperforms the algorithm with no constraint propagation. Furthermore, experimental results show that our algorithm is better than the state-of-the-art alpha-beta search algorithm for large-scale problems. Our future works include developing more efficient algorithms for solving a QCSP by improving the formula of the UCB value calculation and introducing an endgame database. Also, we hope to perform experiments with non-random QCSP instances, such as bin-packing games presented in [12].

Furthermore, our ultimate research goal is to develop an real-time algorithm for a quantified distributed CSP (QDCSP) [2, 14]. A QDCSP is a QCSP in which variables are distributed among agents. In a QDCSP, as in a QCSP, obtaining a
complete plan off-line is intractable. Thus, a team of cooperative agents need to make their decisions in real-time. We hope to extend the algorithm developed in this paper to QDCSP.

Acknowledgments

This research was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A), 20240015, 2008.

References


Fig. 7. Large-scale QCSP against alpha-beta ($n = 50$, $d = 16$)


Agile Asynchronous Backtracking
for Distributed Constraint Satisfaction Problems

Christian Bessiere¹, El Houssine Bouyakhf²,
Younes Mechqrane², and Mohamed Wahbi¹,²

¹ LIRMM/CNRS, University Montpellier II, France
² LIMIARF/FSR, University Mohammed V Agdal, Morocco
{bessiere, wahbi}@lirmm.fr, bouyakhf@fsr.ac.ma
ymechqrane@spsm.ma

Abstract. Asynchronous Backtracking is the standard search procedure for dis-
tributed constraint reasoning. It requires a total ordering on the agents. All poly-
nomial space algorithms proposed so far to improve Asynchronous Backtracking
by reordering agents during search only allow a limited amount of reordering.
In this paper, we propose Agile-ABT, a search procedure that is able to change
the ordering of agents more than previous approaches. This is done via the orig-
inal notion of termination value, a vector of stamps labelling the new orders ex-
changed by agents during search. In Agile-ABT, agents can reorder themselves
as much as they want as long as the termination value decreases as the search
progresses. Our experiments show the good performance of Agile-ABT when
compared to other dynamic reordering techniques.

1 Introduction

Various application problems in distributed artificial intelligence are concerned with
finding a consistent combination of agent actions (e.g., distributed resource allocation
[6], sensor networks [1]). Such problems can be formalized as Distributed Constraint
SATisfaction Problems (DisCSPs). DisCSPs are composed of agents, each owning its
local constraint network. Variables in different agents are connected by constraints.
Agents must assign values to their variables so that all constraints between agents
are satisfied. Several distributed algorithms for solving DisCSPs have been developed,
among which Asynchronous Backtracking (ABT) is the central one [10, 2]. ABT is an
asynchronous algorithm executed autonomously by each agent in the distributed prob-
lem. Agents do not have to wait for decisions of others but they are subject to a total
(priority) order. Each agent tries to find an assignment satisfying the constraints with
what is currently known from higher priority agents. When an agent assigns a value to
its variable, the selected value is sent to lower priority agents. When no value is possible
for a variable, the inconsistency is reported to higher agents in the form of a nogood.
ABT computes a solution (or detects that no solution exists) in a finite time. The to-
tal order is static. Now, it is known from centralized CSPs that adapting the order of
variables dynamically during search drastically fastens the search procedure.

Asynchronous Weak Commitment (AWC) dynamically reorders agents during
search by moving the sender of a nogood higher in the order than the other agents in
the nogood [9]. But AWC requires exponential space for storing nogoods. Silaghi et al. (2001) tried to hybridize ABT with AWC. Abstract agents fulfill the reordering operation to guarantee a finite number of asynchronous reordering operations. In [7], the heuristic of the centralized dynamic backtracking was applied to ABT. However, in both studies, the improvement obtained on ABT was minor. Zivan and Meisels (2006) proposed Dynamic Ordering for Asynchronous Backtracking (ABTDO). When an agent assigns value to its variable, ABTDO can reorder lower priority agents. A new kind of ordering heuristics for ABTDO is presented in [13]. In the best of those heuristics, the agent that generates a nogood is placed between the last and the second last agents in the nogood if its domain size is smaller than that of the agents it passes on the way up.

In this paper, we propose Agile-ABT, an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing before the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Furthermore, that agent receiving the backtrack message, called the backtracking target, is not necessarily the agent with the lowest priority within the conflicting agents in the current order. The principle of Agile-ABT is built on termination values exchanged by agents during search. A termination value is a tuple of positive integers attached to an order. Each positive integer in the tuple represents the expected current domain size of the agent in that position in the order. Orders are changed by agents without any global control so that the termination value decreases lexicographically as the search progresses. Since, a domain size can never be negative, termination values cannot decrease indefinitely. An agent informs the others of a new order by sending them its new order and its new termination value. When an agent compares two contradictory orders, it keeps the order associated with the smallest termination value.

The rest of the paper is organized as follows. Section 2 recalls basic definitions. Section 3 describes the concepts needed to select new orders that decrease the termination value. We give the details of our algorithm in Section 4 and we prove it in Section 5. An extensive experimental evaluation is given in Section 6. Section 7 concludes the paper.

2 Preliminaries

The Distributed Constraint Satisfaction Problem (DisCSP) has been formalized in [10] as a tuple $(A, X', D, C)$, where $A$ is a set of agents, $X'$ is a set of variables $\{x_1, \ldots, x_n\}$, where each variable $x_i$ is controlled by one agent in $A$. $D = \{D_1, \ldots, D_n\}$ is a set of domains, where $D_i$ is a finite set of values to which variable $x_i$ may be assigned. The initial domain size of a variable $x_i$ is denoted by $d^0_i$. $C$ is a set of binary constraints that specify the combinations of values allowed for the two variables they involve. A constraint $c_{ik} \in C$ between two variables $x_i$ and $x_k$ is a subset of the Cartesian product $D_i \times D_k$.

For simplicity purposes, we consider a restricted version of DisCSP where each agent controls exactly one variable. We use the terms agent and variable interchangeably and we identify the agent ID with its variable index. All agents maintain their own
counter, and increment it whenever they change their value. The current value of the counter tags each generated assignment.

**Definition 1.** An assignment for an agent \( A_i \in \mathcal{A} \) is a tuple \((x_i, v_i, t_i)\), where \( v_i \) is a value from the domain of \( x_i \) and \( t_i \) is the tag value. When comparing two assignments, the most up to date is the one with the highest tag \( t_i \). Two sets of assignments \( \{(x_{i1}, v_{i1}, t_{i1}), \ldots, (x_{ik}, v_{ik}, t_{ik})\} \) and \( \{(x_{j1}, v_{j1}, t_{j1}), \ldots, (x_{jq}, v_{jq}, t_{jq})\} \) are coherent if every common variable is assigned the same value in both sets.

\( A_i \) is allowed to store a unique order denoted by \( o_i \). Agents appearing before \( A_i \) in \( o_i \) are the higher agents (predecessors) denoted by \( \text{Pred}(A_i) \) and conversely the lower agents (successors) \( \text{Succ}(A_i) \) are agents appearing after \( A_i \).

**Definition 2.** The AgentView of an agent \( A_i \) is an array containing the most up to date assignments of \( \text{Pred}(A_i) \).

Agents can infer inconsistent sets of assignments, called **nogoods**. A nogood can be represented as an implication. There are clearly many different ways of representing a given nogood as an implication. For example, \( \neg([x_1=v_1) \land \cdots \land (x_k=v_k)] \) is logically equivalent to \([x_2=v_2) \land \cdots \land (x_k=v_k)] \rightarrow (x_1 \neq v_1) \). When a nogood is represented as an implication, the left hand side (\( \text{lhs} \)) and the right hand side (\( \text{rhs} \)) are defined from the position of \( \rightarrow \). A nogood is compatible with an order \( o_i \) if all agents in \( \text{lhs}(\text{nogood}) \) appear before \( \text{rhs}(\text{nogood}) \) in \( o_i \).

The current domain of \( x_i \) is the set of values \( v \in D_i \) such that \( x_i \neq v \) does not appear in any of the right hand sides of the nogoods stored by \( A_i \). Each agent keeps only one nogood per removed value. The size of the current domain of \( A_i \) is denoted by \( d_i \).

### 3 Introductory Material

Before presenting Agile-ABT, we need to introduce new notions and to present some key subfunctions.

#### 3.1 Reordering details

There is one major issue to be solved for allowing agents to asynchronously propose new orders: The agents must be able to coherently decide which order to select. We propose that the priority between the different orders is based on termination values. Informally, if \( o_i = [A_1, \ldots, A_n] \) is the current order known by an agent \( A_i \), then the tuple of domain sizes \([d_1, \ldots, d_n]\) is the termination value of \( o_i \) on \( A_i \). To build termination values, agents need to exchange explanations.

**Definition 3.** An explanation \( e_j \) is an expression of the form \( \text{lhs}(e_j) \rightarrow d_j \), where \( \text{lhs}(e_j) \) is the conjunction of the left hand sides of all nogoods stored by \( A_j \) as justifications of value removals, and \( d_j \) is the number of values not pruned by nogoods in the domain of \( A_j \). \( d_j \) is also denoted by \( \text{rhs}(e_j) \).
Each time an agent communicates its assignment to other agents (by sending them an ok? message), it inserts its explanation in the ok? message for allowing other agents to build their termination value.

The variables in the left hand side of an explanation $e_j$ must precede the variable $x_j$ in the order because the assignments of these variables have been used to determine the current domain of $x_j$. An explanation $e_j$ induces ordering constraints, called safety conditions in [4].

**Definition 4.** A safety condition is an assertion $x_k \prec x_j$. Given an explanation $e_j$, $S(e_j)$ is the set of safety conditions induced by $e_j$, where $S(e_j) = \{(x_k \prec x_j) \mid x_k \in \text{lhs}(e_j)\}$.

An explanation $e_j$ is compatible with an order $o$ if all variables in $\text{lhs}(e_j)$ appear before $x_j$ in $o$. Each agent $A_i$ stores a set $E_i$ of explanations sent by other agents. During search, $E_i$ is updated to remove explanations that are no longer valid.

**Definition 5.** An explanation $e_j$ in $E_i$ is valid on agent $A_i$ if it is compatible with the current order $o_i$ and $\text{lhs}(e_j)$ is coherent with the AgentView of $A_i$.

When $E_i$ contains an explanation $e_j$ associated with $A_j$, $A_i$ uses this explanation to justify the size of the current domain of $A_j$. Otherwise, $A_i$ assumes that the size of the current domain of $A_j$ is equal to $d^0_j$. The termination value depends on the order and the set of explanations.

**Definition 6.** Let $E_i$ be the set of explanations stored by $A_i$, $o$ be an order on the agents such that every explanation in $E_i$ is compatible with $o$, and $o(k)$ be such that $A(o(k))$ is the $k$th agent in $o$. The termination value $TV(E_i, o)$ is the tuple $[tv^1, \ldots, tv^n]$, where $tv^k = \text{rhs}(e_{o(k)})$ if $e_{o(k)} \in E_i$, otherwise, $tv^k = d^0_{o(k)}$.

In Agile-ABT, an order is always associated with a termination value. When comparing two orders the strongest order is that associated with the lexicographically smallest termination value. In case of ties, we use the lexicographic order on agents IDs, the smaller being the stronger.

### 3.2 The backtracking target

When all values of an agent $A_i$ are ruled out by nogoods, these nogoods are resolved, producing a new nogood $ng$. $ng$ is the conjunction of $\text{lhs}$ of all nogoods stored by $A_i$. If $ng$ is empty, then the inconsistency is proved. Otherwise, one of the conflicting agents must change its value. In standard ABT, the agent that has the lowest priority must change its value. Agile-ABT overcomes this restriction by allowing $A_i$ to select with great freedom the target agent $A_k$ who must change its value (i.e., the variable to place in the right hand side of $ng$). The only restriction to place a variable $x_k$ in the right hand side of $ng$ is to find an order $o'$ such that $TV(up_E, o')$ is lexicographically smaller than the termination value associated with the current order of $A_i$. $up_E$ is obtained by updating $E_i$ after placing $x_k$ in $\text{rhs}(ng)$. 

---

19
the nogood \((ng)\) and will inform all other agents. The agent will replace its current order and termination value by those just computed, such that the associated termination value is smaller than that of the current order. If so, we aim at minimizing the termination value, function \(\text{computeOrder}\) works by determining, at each iteration \(p\), the set \(\text{Roots}\) of vertices that have no predecessor. As we aim at minimizing the termination value, function \(\text{computeOrder}\) selects the vertex \(x_j\) in \(\text{Roots}\) that has the smallest domain size. This vertex is placed at the \(p\)th position and removed from \(G\). Finally, \(p\) is incremented and all outgoing edges from \(x_j\) are removed from \(G\).

Having proposed an algorithm that determines an order with small termination value for a given backtracking target \(x_k\), one needs to know how to choose this

```python
function updateExplanations \((E_k, ng, x_k)\)
1. \(up_{\_E} \leftarrow E_k\); \(setRhs\) \((ng, x_k)\);
2. remove each \(e_j \in up_{\_E}\) such that \(x_k \in lhs\) \((e_j)\);
3. if \((e_k \notin up_{\_E})\) then \(setLhs\) \((e_k, 0)\); \(setRhs\) \((e_k, d_0^k)\); add \(e_k\) to \(up_{\_E}\);
4. \(setLhs\) \((e_k', lhs\) \((e_k) \cup lhs\) \((ng)\)); \(setRhs\) \((e_k', rhs\) \((e_k) - 1)\);
5. replace \(e_k\) by \(e_k'\);
6. return \(up_{\_E}\);
```

Function \(\text{updateExplanations}\) takes as arguments the set \(E_i\), the nogood \(ng\) and the variable \(x_k\) to place in the \(rhs\) of \(ng\). \(\text{updateExplanations}\) removes all explanations that are no longer coherent after placing \(x_k\) in the right hand side of \(ng\). It updates the explanation of agent \(A_k\) stored in \(A_i\) and it returns a set of explanations \(up_{\_E}\).

This function does not create cycles in the set of safety conditions \(S(up_{\_E})\) if \(S(E_i)\) is acyclic. Indeed, all the explanations added or removed from \(S(E_i)\) to obtain \(S(up_{\_E})\) contain \(x_k\). Hence, if \(S(up_{\_E})\) contains cycles, all these cycles should contain \(x_k\). However, there does not exist any safety condition of the form \(x_k \prec x_j\) in \(S(up_{\_E})\) because all of these explanations have been removed in line 2. Thus, \(S(up_{\_E})\) cannot be cyclic. As we will show in Section 4, the updates performed by \(A_i\) ensure that \(S(E_i)\) always remains acyclic. As a result, \(S(up_{\_E})\) is acyclic as well, and it can be represented by a directed acyclic graph \(G = (N, U)\) such that \(N = \{x_1, \ldots, x_n\}\) is the set of nodes and \(U\) is the set of directed edges. An edge \((j, l)\) \(\in U\) if \((x_j \prec x_l) \in S(up_{\_E})\). Thus, any topological sort of \(G\) is an order that agrees with the safety conditions induced by \(up_{\_E}\).

### 3.3 Decreasing termination values

Termination of Agile-ABT is based on the fact that the termination values associated with orders selected by agents decrease as search progresses. To speed up the search, Agile-ABT is written so that agents decrease termination values whenever they can. When an agent resolves its nogoods, it checks whether it can find a new order of agents such that the associated termination value is smaller than that of the current order. If so, the agent will replace its current order and termination value by those just computed, and will inform all other agents.

Assume that after resolving its nogoods, an agent \(A_i\), decides to place \(x_k\) in the \(rhs\) of the nogood \((ng)\) produced by the resolution and let \(up_{\_E} = \text{updateExplanations}\) \((E_i, ng, x_k)\). The function \(\text{computeOrder}\) takes as parameter the set \(up_{\_E}\) and returns an order \(up_{\_O}\) compatible with the partial ordering induced by \(up_{\_E}\). Let \(G\) be the acyclic directed graph associated with \(up_{\_E}\). The function \(\text{computeOrder}\) works by determining, at each iteration \(p\), the set \(\text{Roots}\) of vertices that have no predecessor. As we aim at minimizing the termination value, function \(\text{computeOrder}\) selects the vertex \(x_j\) in \(\text{Roots}\) that has the smallest domain size. This vertex is placed at the \(p\)th position and removed from \(G\). Finally, \(p\) is incremented and all outgoing edges from \(x_j\) are removed from \(G\).
function computeOrder(up\_E)
7. $G = (N, U)$ is the acyclic graph associated to $up\_E$;
8. $p \leftarrow 1$; $o$ is an array of length $n$;
9. while $G \neq \emptyset$ do
10. $Roots \leftarrow \{x_j \in N \mid x_j$ has no incoming edges$\}$;
11. $o[p] \leftarrow x_j$ such that $d_j = \min\{d_k \mid x_k \in Roots\}$;
12. remove $x_j$ from $G$; $p \leftarrow p + 1$;
13. return $o$;

variable to obtain an order decreasing more the termination value. The function chooseVariableOrder iterates through all variables $x_k$ included in the nogood, computes a new order and termination value with $x_k$ as the target (lines 16–17), and stores the target and the associated order if it is the strongest order found so far (lines 18–19). Finally, the information corresponding to the strongest order is returned.

function chooseVariableOrder($E_i, ng$)
14. $o' \leftarrow o_i$; $TV' \leftarrow TV_i$; $E' \leftarrow nil$; $x' \leftarrow nil$;
15. for each $x_k \in ng$ do
16. $up\_E \leftarrow$ updateExplanations($E_i, ng, x_k$) ;
17. $up\omega \leftarrow$ computeOrder($up\_E$); $up\_TV \leftarrow TV(up\_E, up\omega)$;
18. if ($up\_TV$ is smaller than $TV'$) then
19. $x' \leftarrow x_k$; $o' \leftarrow up\omega$; $TV' \leftarrow up\_TV$; $E' \leftarrow up\_E$;
20. return $(x', o', TV', E')$;

4 The Algorithm

Each agent keeps some amount of local information about the global search, namely an AgentView, a NogoodStore, a set of explanations ($E_i$), a current order ($o_i$) and a termination value ($TV_i$). Agile-ABT allows the following types of messages (where $A_i$ is the sender):

– ok? message is sent by $A_i$ to lower agents to ask whether a chosen value is acceptable. Besides the chosen value, the ok? message contains an explanation $e_i$ which communicates the current domain size of $A_i$. An ok? message also contains the current order $o_i$ and the current termination value $TV_i$ stored by $A_i$.

– ngd message is sent by $A_i$ when all its values are ruled out by its NogoodStore. This message contains a nogood, as well as $o_i$ and $TV_i$.

– order message is sent to propose a new order. This message includes the order $o_i$ proposed by $A_i$ accompanied by the termination value $TV_i$.

Agile-ABT (Figure 1 and 2) is executed on every agent. After initialization, each agent assigns a value and informs lower priority agents of its decision (CheckAgentView call, line 22) by sending ok? messages. Then, a loop considers the
procedure Agile-ABT ()
21. $t_i \leftarrow 0; TV_i \leftarrow [\infty, \infty, \ldots, \infty]; end \leftarrow \text{false}; v_i \leftarrow \text{empty};$
22. CheckAgentView () ;
23. while (~end) do
24. $msg \leftarrow \text{getMsg}() ;$
25. switch (msg.type) do
26. ok? : ProcessInfo (msg);
27. order : ProcessOrder (msg);
28. ngd : ResolveConflict (msg);
29.stp : end $\leftarrow \text{true};$

procedure ProcessInfo (msg)
30. CheckOrder (msg.Order , msg.TV) ;
31. UpdateAgentView (msg.Assig $\cup$ lhs (msg.Exp)) ;
32. if msg.Exp is valid then add (msg.Exp, $E$ ) ;
33. CheckAgentView () ;

procedure ProcessOrder (msg)
34. CheckOrder (msg.Order , msg.TV) ;
35. CheckAgentView () ;

procedure ResolveConflict (msg)
36. CheckOrder (msg.Order , msg.TV) ;
37. UpdateAgentView (msg.Assig $\cup$ lhs (msg.Nogood)) ;
38. if Coherent (msg.Nogood, AgentView $\cup$ $x_i=v_i$ ) and Compatible (msg.Nogood, $o_i$ )
then
39. add (msg.Nogood,NogoodStore) ; $v_i \leftarrow \text{empty};$
40. CheckAgentView () ;
41. else if rhs (msg.Nogood) = $v_i$ then
42. sendMsg: ok?($v_i, e_i, o_i, TV_i$) to msg.Sender ;

procedure CheckOrder (o, TV)
43. if o is stronger than $o_i$ then $o_i \leftarrow o; TV_i \leftarrow TV$;
44. remove nogoods and explanations incompatible with $o_i$;

procedure CheckAgentView ()
45. if ~Consistent ($v_i,AgentView$) then
46. $v_i \leftarrow \text{ChooseValue}() ;$
47. if ($v_i$) then sendMsg: ok?($v_i, e_i, o_i, TV_i$) to Succ($A_i$);
48. else Backtrack () ;
49. else if ($o_i$ was modified) then
50. sendMsg: ok?($v_i, e_i, o_i, TV_i$) to Succ($A_i$);

procedure UpdateAgentView (Assignments)
51. for each var $\in$ Assignments do
52. if Assignments[var].c > AgentView [var].c then
53. AgentView [var] $\leftarrow$ Assignments[var];
54. remove nogoods and explanations incoherent with AgentView;

Fig. 1. The Agile-ABT algorithm (Part 1).
procedure Backtrack()
55. $ng \leftarrow \text{solve}(\text{NogoodStore})$
56. if $(ng = \text{empty})$ then $\text{end} \leftarrow \text{true}; \text{sendMsg} := \text{stp}($system$)$
57. $(x_k, o', TV', E') \leftarrow \text{chooseVariableOrder}(E_i, ng)$
58. if $(TV' \text{ is smaller than TV}_i)$ then
59. $TV_i \leftarrow TV'; o_i \leftarrow o'; E_i \leftarrow E'$
60. setRhs $(ng, x_k)$;
61. $\text{sendMsg} := \text{ngd}(ng, o_i, TV)$ to $A_k$
62. remove $e_k$ from $E_i$
63. $\text{broadcastMsg} := \text{order}(o_i, TV)$;
64. else
65. setRhs $(ng, x_k)$;
66. $\text{sendMsg} := \text{ngd}(ng, o_i, TV)$ to $A_k$
67. $\text{UpdateAgentView}(x_k \leftarrow \text{unknown})$
68. $\text{CheckAgentView}()$

function ChooseValue()
69. for each $(v \in D_i \text{ not eliminated by NogoodStore})$ do
70. if $(\exists x_j \in \text{AgentView} \text{ such that } \neg\text{Consistent}(v, x_j))$ then
71. add $(x_j = v_j \Rightarrow x_i \neq v, \text{NogoodStore})$
72. if $(D_i = \emptyset)$ then return (empty)
73. else $t_i \leftarrow t_i + 1$ return $(v)$; /* $v \in D_i$ */

Fig. 2. The Agile-ABT algorithm (Part 2).

reception of the possible message types. If no message is traveling through the network, the state of quiescence is detected by a specialized algorithm [5], and a global solution is announced. The solution is given by the current variables’ assignments.

When an agent $A_i$ receives a message (of any type), it checks if the order included in the received message is stronger than its current order $o_i$ (CheckOrder call, lines 30, 34 and 36). If it is the case, $A_i$ replaces $o_i$ and $TV_i$ by those newly received (line 43). The nogoods and explanations that are no longer compatible with $o_i$ are removed to ensure that $S(E_i)$ remains acyclic (line 44).

If the message was an ok? message, the AgentView of $A_i$ is updated to include the new assignments (UpdateAgentView call, line 31). Beside the assignment of the sender, $A_i$ also takes newer assignments contained in the left hand side of the explanation included in the received ok? message to update its AgentView. Afterwards, the nogoods and the explanations that are no longer coherent with AgentView are removed (UpdateAgentView line 54). Then, if the explanation in the received message is valid, $A_i$ updates the set of explanations by storing the newly received explanation. Next, $A_i$ calls the procedure CheckAgentView (line 33).

When receiving an order message, $A_i$ processes the new order (CheckOrder) and calls CheckAgentView (line 35).
When $A_i$ receives a $\text{ngd}$ message, it calls $\text{CheckOrder}$ and $\text{UpdateAgentView}$ (lines 36 and 37). The nogood contained in the message is accepted if it is coherent with the AgentView and the assignment of $x_i$ and compatible with the current order of $A_i$. Otherwise, the nogood is discarded and an $\text{ok}$? message is sent to the sender as in ABT (lines 41 and 42). When the nogood is accepted, it is stored, acting as justification for removing the current value of $A_i$ (line 39). A new value consistent with the AgentView is searched ($\text{CheckAgentView}$ call, line 40).

The procedure $\text{CheckAgentView}$ checks if the current value $v_i$ is consistent with the AgentView. If $v_i$ is consistent, $A_i$ checks if $o_i$ was modified (line 49). If so, $A_i$ must send its assignment to lower priority agents through $\text{ok}$? messages. If $v_i$ is not consistent with its AgentView, $A_i$ tries to find a consistent value ($\text{ChooseValue}$ call, line 46). In this process, some values of $A_i$ may appear as inconsistent. In this case, the nogoods justifying their removal are added to the NogoodStore (line 71 of function $\text{ChooseValue}$). If a new consistent value is found, an explanation $e_i$ is built and the new assignment is notified to the lower priority agents of $A_i$ through $\text{ok}$? messages (line 47). Otherwise, every value of $A_i$ is forbidden by the NogoodStore and $A_i$ has to backtrack ($\text{Backtrack}$ call, line 48).

In procedure $\text{Backtrack}$, $A_i$ resolves its nogoods, deriving a new nogood ($ng$). If $ng$ is empty, the problem has no solution. $A_i$ terminates execution after sending a $\text{stp}$ message (line 56). Otherwise, one of the agents included in $ng$ must change its value. The function $\text{chooseVariableOrder}$ selects the variable to be changed ($x_k$) and a new order ($o'$) such that the new termination value $TV'$ is as small as possible. If $TV'$ is smaller than that stored by $A_i$, the current order and the current termination value are replaced by $o'$ and $TV'$ and $A_i$ updates its explanations by that returned by $\text{chooseVariableOrder}$ (line 59). Then, a $\text{ngd}$ message is sent to the agent $A_k$ owner of $x_k$ (line 61). Then, $e_k$ is removed from $E_i$ since $A_k$ will probably change its explanation after receiving the nogood (line 62). Afterwards, $A_i$ sends an $\text{order}$ message to all other agents (line 63). When $TV'$ is not smaller than the current termination value, $A_i$ cannot propose a new order and the variable to be changed ($x_k$) is the variable that has the lowest priority according to the current order of $A_i$ (lines 65 and 66). Next, the assignment of $x_k$ (the target of the backtrack) is removed from the AgentView of $A_i$ (line 67). Finally, the search is continued by calling the procedure $\text{CheckAgentView}$ (line 68).

5 Correctness and Complexity

We demonstrate that Agile-ABT is sound, complete and terminates, with a polynomial space complexity.

**Theorem 1.** Agile-ABT requires $O(nd + n^2)$ space per agent.

**Theorem 2.** Agile-ABT is sound.

**Proof.** (Sketch) When the state of quiescence is reached, all agents necessarily know the order $o$ that is the strongest ever computed. In addition, all agents know the most up to date assignments of all their predecessors in $o$. Thus, any constraint $c_{ik}$ between
agents $A_i$ and $A_k$ has been successfully checked by the agent with lowest priority in $o$. Otherwise that agent would have tried to change its value and would have either sent an ok? or a ngd message, breaking the quiescence.

**Theorem 3.** Agile-ABT is complete.

*Proof.* All nogoods are generated by logical inferences from existing constraints. Therefore, an empty nogood cannot be inferred if a solution exists.

The proof of termination is built on lemmas 1 and 2.

**Lemma 1.** For every agent $A_i$, while no solution is found and the inconsistency of the problem is not proved, the termination value stored by $A_i$ decreases after a finite amount of time.

*Proof.* (Sketch) If an agent gets stuck a sufficiently long time with the same termination value, all agents will eventually have that same termination value and the order $o$ to which it was attached. At this point, Agile-ABT works exactly like ABT, which is complete and terminates. Thus, either a solution is found or the first agent in the current order, $A_{o(1)}$, will receive a nogood with empty $lhs$ that prunes one of its remaining values. As soon as $A_{o(1)}$ has sent its new domain size (smaller than $tv^1$) to the lower agents in $o$, any backtracking agent will generate a smaller termination value.

**Lemma 2.** For any termination value $TV = [tv^1, \ldots, tv^n]$ generated by an agent, we have $tv^j \geq 0, \forall j \in 1..n$

*Proof.* (Sketch) All explanations $e_k$ stored by an agent $A_i$ have $rhs(e_k) \geq 1$ because it represents the current domain size of $A_k$. Now, termination values are built with $rhs$ of explanations. The only case where a $tv^j$ can be zero (line 4) is when $A_{o(j)}$ is selected by $A_i$ to be the backtracking target, and in such a case, the explanation $e_{o(j)}$ is removed just after sending the nogood to $A_{o(j)}$ (line 62). Hence, $A_i$ never stores an explanation $e_k$ with $rhs(e_k) = 0$ and cannot produce a termination value with a negative element.

**Theorem 4.** Agile-ABT terminates.

*Proof.* Direct from Lemmas 1 and 2.

6 Experimental Results

We compared Agile-ABT to ABT, ABTDO, and ABTDO with retroactive heuristics. All experiments were performed on the DisChoco 2.0 [3] platform,\(^1\) in which agents are simulated by Java threads that communicate only through message passing. We evaluate the performance of the algorithms by communication load and computation effort. Communication load is measured by the total number of messages exchanged among agents during algorithm execution ($\#msg$), including termination detection (system

\(^1\) [http://www.lirmm.fr/coconut/dischoco/](http://www.lirmm.fr/coconut/dischoco/)
messages). Computation effort is measured by an adaptation of the number of non-concurrent constraint checks ($\#nccc$) [12] where we also count nogood checks to be closer to the actual computational effort.

For ABT, we implemented the standard version where we use counters for tagging assignments. For ABTDO [11], we implemented the best version, using the nogood-triggered heuristic where the receiver of a nogood moves the sender to be in front of all other lower priority agents (denoted by ABTDO-ng). For ABTDO with retroactive heuristics [13], we implemented the best version, in which a nogood generator moves itself to be in a higher position between the last and the second last agents in the generated nogood. However, it moves before an agent only if its current domain is smaller than the domain of that agent (denoted by ABTDO-Retro).

The algorithms are tested on uniform binary random DisCSPs that are characterized by $\langle n, d, p_1, p_2 \rangle$ where $n$ is the number of agents/variables, $d$ the number of values per variable, $p_1$ the network connectivity defined as the ratio of existing binary constraints, and $p_2$ the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of problems: sparse problems $\langle 20, 10, 0.20, p_2 \rangle$ and dense problems $\langle 20, 10, 0.70, p_2 \rangle$. We vary the tightness $p_2$ from 0.10 to 0.90 by steps of 0.10. For each pair of fixed density and tightness $(p_1, p_2)$ we generated 25 instances, solved 4 times each. We report average over the 100 runs.

Figure 3 presents the results on the sparse instances ($p_1 = 0.20$). In terms of computational effort ($\#ncccs$) (top of figure 3), ABT is the less efficient algorithm. ABTDO-ng improves ABT by a large scale and ABTDO-Retro is more efficient than ABTDO-ng. These findings are similar to those reported in [13]. Agile-ABT outperforms all these algorithms, suggesting that on sparse problems, the more sophisticated the algorithm is, the better it is. Regarding the number of exchanged messages ($\#msg$) (bottom of figure 3), the faster resolution may not translate in an overall communication load reduction. ABT requires less messages than ABTDO-ng and ABTDO-Retro. On the contrary, Agile-ABT is the algorithm that requires the smallest number of messages despite its extra messages sent by agents to notify the others of a new ordering, even compared to ABT. This is not only because Agile-ABT terminates faster than the other algorithms (see $\#ncccs$). A second reason is that Agile-ABT is more parsimonious than ABTDO algorithms in proposing new orders. Termination values seem to focus changes on those which will pay off.

Figure 4 presents the results on the dense instances ($p_1 = 0.70$). Some differences appear compared to sparse problems. Concerning $\#ncccs$ (top of figure 4), ABTDO algorithms deteriorate compared to ABT. However, Agile-ABT still outperforms all these algorithms. Regarding communication load ($\#msg$) (bottom of figure 4), ABTDO-ng and ABTDO-Retro show the same bad performance as in sparse problems. On the contrary, Agile-ABT is approximately as good as ABT. This confirms its good behavior observed on sparse problems.

From these experiments we can conclude that Agile-ABT outperforms other algorithms in terms of computation load ($\#ncccs$) on all types of problems tested. Concerning communication load ($\#msg$), Agile-ABT is more robust than other versions of ABT with dynamic agent ordering. As opposed to them, it is always better than or as good as standard ABT on difficult problems.
Fig. 3. Total \#msg exchanged and \#ncccs performed on sparse graphs ($p_1 = 0.20$).

7 Conclusion

We have proposed Agile-ABT, an algorithm that is able to change the ordering of agents more agilely than all previous approaches. Thanks to the original concept of termination value, Agile-ABT is able to choose a backtracking target that is not necessarily the agent with the current lowest priority within the conflicting agents. Furthermore, the ordering of agents appearing before the backtracking target can be changed. These interesting features are unusual for an algorithm with polynomial space complexity. Our experiments confirm the significance of these features.
Fig. 4. Total $\#\text{msg}$ exchanged and $\#\text{ncccs}$ performed on dense graphs ($p_1 = 0.70$).
Bibliography

The Distributed Constraints (DisCo) Simulation Tool *

A. Grubshtein, N. Herschhorn, A. Netzer, G. Rapaport, G. Yafe and A. Meisels

Ben-Gurion University of the Negev,
Department of Computer Science,
P.O.B 653, Be’er Sheva, 84105, Israel
{alongrub,herschho,netzerar,guyrap,guyafe,am}@cs.bgu.ac.il

Abstract. A simulation tool for distributed constraints search and optimization algorithms is presented. The DisCo SimTool is the product of several generations of simulators that served all researchers of the distributed constraints reasoning (DCR) group at BGU. It includes all needed abstractions for implementing all existing DCSP and DCOP algorithms. From Synchronous BackTracking (SBT) to asynchronous algorithms like ABT and concurrent algorithms like ConcDB. From pseudo-tree algorithms like ADOPT, to complex structured ones like OptAPO. DisCo includes all the needed infrastructure for designing, implementing, and testing DCSP and DCOP algorithms. An asynchronous Mailer, a Statistical package that includes a variety of distributed run-time and network load measures, and a complete GUI that enables the design and implementation of comparative evaluations of all algorithms. Last but not least, DisCo includes also a Visualization Module that presents visually multiple features of DCOP algorithms during their run-time.

1 Introduction

The BGU Distributed Constraints Reasoning research group has been studying algorithms and variants of DCR problems over the last decade. During these years, different members of the group have been independently developing frameworks for designing, running, and testing a variety of DCR algorithms. The Distributed Constraints (DisCo) Simulation Tool described in this article is a culmination of all of these efforts and design decisions during the last 7 years. Variants of the BGU-DCR Simulator have been maintained for research on DCSP algorithms (AFC, ABT, ConcBT [10, 18, 2, 20, 21]), on DCOP algorithms (ADOPT, AFB [12, 5]), on complex algorithms (DPOP, APO [13, 6]), on asymmetric DCSPs & DCOPs [3, 7] and on the behavior of multiple algorithms under message delays [22, 19]. We aim to make this tool the basis for future algorithms development within our group and to enable any interested researcher to use it freely. As the tool has now reached a certain level of maturity, we have decided to make it available for downloading to the worldwide DCR community.

* This work was supported by the Lynn and William Frankel Center for Computer Science, and of the Paul Ivanier Center for Robotics.
The Simulation Tool that will be described in the present paper strives to achieve a complete, simple, measurable and reliable framework, to investigate existing algorithms and to support research for new ones. To extend cross-platform compatibility, we chose Java as our implementation language. Java libraries (such as JUNG) are used to enhance the tool’s abilities.

The framework of the DisCo SimTool includes all that is required for implementing all existing DCR algorithms and for performing extensive comparative evaluation of these algorithms. The framework allows a researcher to express a DCR algorithm by implementing the send/receive message handlers of the algorithm’s agent. The underlying mailer system for these messages, along with statistics gathering services and ADTs for the problem formulations are supplied by the framework. The framework includes a GUI and a visualization tool, which will be described in detail.

Simplicity and ease of implementation were a major concern during the DisCo design phase, and a great effort was made to meet these requirements. All services offered by the framework are available for new agent classes by inheriting from an existing, abstract agent class.

The DisCo SimTool includes several performance metrics, such as run-time in the form of Non-Concurrent Constraints Checks (NCCCs) [22, 11], network load measured as the total number (or volume) of messages sent, and others. Both run-time and network load which are the standard performance measures of distributed algorithms [9] have several implemented versions in DisCo. From NCCCs, which are suitable for ABT, AFC, ConcBT, to non-concurrent logical operations which are more suitable for DPOP and APO.

New metrics can be easily aggregated at chosen points in the code, and be used later, either as file output or by one of the visualization modules. This allows flexibility in the definition of performance metrics. Instead of limiting the user to a specific set of metrics, the statistics module can be easily configured to service new types of statistics. The implemented statistics module allows customized statistics aggregation (e.g. NCCCs versus the number of messages). This enables users to review how and where in the code statistics are gathered. Reliability in comparing different algorithms can be achieved and assured by collecting measurements in any desired fashion.

Code constructs exist for defining which parameters in an algorithm’s execution should be configurable by the user and a GUI component is dynamically generated for these parameters. This enables GUI support for newly-implemented algorithms without having to updated the GUI’s code.

A checking module is also available. This module examines the validity of the results returned by a distributed algorithm for a given DCR problem. It implements fast algorithms which search for solutions on the centralized version of the problem at hand and issue a warning when inconsistencies are detected. Such functionality is useful during the implementation and debugging phases of new DCR algorithms.

The DisCo SimTool is designed with extensibility in mind: the design is modular and allows independent development and upgrade of alternative im-
plementations to most of the modules. Customized problem generators can be implemented and allows for the generation of structured problems or domain-specific problems. The message delay policy of the mailer system can be altered as well, allowing users to test the same algorithms under different network conditions. New algorithms can be downloaded and plugged-in to the tool in the form of JARs. New visualization modules can also be downloaded and plugged-in or implemented according to specific needs. These tools are independent of the simulator’s core code and can be further developed and upgraded without knowledge of its workings. The principal guideline is exposing the programmer to as little code as possible, thus allowing users to focus on their newly-written code only.

Visualization takes a significant part in our new tool. We have come to believe that by visualizing different aspects of distributed search algorithms, one can attain new insights on the behavior of these algorithms. These insights may help drive new variants of search algorithms and different heuristics for these algorithms. Visualizations is also a helpful debugging tool that can be used during the implementation process of new algorithms, and can serve as an educational method which dynamically displays an algorithm’s progress.

In pursue of these goals, we have begin to gradually expose this framework to students in the field of Constraints Processing, encouraging them to implement their algorithms on the Simulator’s framework. The tool now seems mature enough to be exposed to the general audience, and enable implementation of new algorithms by researchers that are unfamiliar with its history and design. The following sections will describe DisCo’s principles and general structural design in an attempt to facilitate understanding and speed the implementation process of new users.

2 Main Components

While other Frameworks for DCR Simulation and Visualization were released along the years [8, 17, 16], each of us kept using the modules and implementations he or she individually created. Needs to integrate efforts within the group arose: for instance, when we wanted to compare experimental results with each other. Over the time we began to integrate existing parts of code written by various group members a bigger framework which satisfied our needs. This is the environment our tool has emerged from: a group of individuals preferring a minimal set of features which can be easily integrated into a larger simulation system. We have tried to adhere to these principles when further developing the tool.

The following services, offered by the framework, will be briefly discussed. A complete implementation of an agent that runs the Synchronous BackTracking algorithm (SBT) can be found in the first appendix, and the lines of code appearing in this section are taken from there.
1. **Mailer System**
   The distributed environment is based on message-passing between agents. When implementing a new algorithm, this adds an overhead of creating a thread-safe mailer system for the algorithm to use. Our framework includes such a mailer system which has been extensively tested. The API for the mailer system is very simple and does not involve adding many lines of code to the algorithm’s core logic. For instance: this is how an agent pulls a message from its mailbox, and sends a new message to the next agent.

```
AbstractMessage message_from_mailbox = getMessageFromMailBox();
if (message_from_mailbox instanceof SimulationFinished) {
    // ... handling a received message.
    // Finally responding by sending a message to the next agent.
    sendMessage(new SBT_Message(id, id + 1, ltc, cpa));
}
```

Please note that SBT_Message is a subclass of the general-purpose AbstractMessage, which can be extended with any data an algorithm requires to carry atop of the messages.

2. **Problem Representation and Generation**
   Problems can be generated based on given parameters or by reading given random problem files. An abstract problem class has been extended onto specific implementations of CSP, COP and ADCOP problem classes, allowing implementations of algorithms for these types of problems. The Problem interface allows querying whether two agents are constrained, in which variables and in which costs.

3. **Statistics Database**
   A Statistics Database that can handle inputs from multiple sources is included in the framework. The database allows aggregation of data from agents. This data can be dumped into files recording the execution, or polled by the visualization modules and displayed in real time. The supported statistics are unbounded, as they are controlled by the user. Below is a code sample which shows how a statistic is declared and gathered by the database.

```
// During an agent’s initialization, the statistics is registered:
registerAgentStatisticsObjectService(new StatisticsObject("←SomeMeasurement"));

// Registered statistics can be put anywhere in the code, reflecting←
// the measurement’s semantics:
accumulateAgentStatistics("SomeMeasurement", 5); // Increment by n.

// The Visualization Driver can poll a registered statistic:
sdb.getStatistics("SomeMeasurement", 3); // Collect statistic "←
SomeMeasurement" of agent #3.
```

The Visualization Module’s interface allows querying which statistical measurements are of interest to it; if these are supported by the algorithm then they can be collected during runtime and used in visualizing the algorithm’s execution.
4. Extensibility
Algorithms, Problem Generators (for structured or domain-specific problems), Visualization Modules and accompanying GUI panels - all of these can be dynamically loaded from JARs, as long as they comply with the supplied interfaces. This allows users of the framework to exchange algorithms or visualizations with other users, including GUI for the new features.

3 The Graphical User Interface

We have identified a GUI front-end as a key ingredient in a successful DisCo tool. An experiment involving DCR algorithms is a combination of problem generation, algorithms to handle these problems, and statistics-gathering settings. While each of these elements may require its own set of settings, a common use-case involves the definition of sets of problems or algorithms and recurrent executions (same problems on different algorithms, same problems on same algorithms with different settings, having the environment parameters such as message delay policy change, et cetera).

All of these options are embedded in the GUI for the framework. We believe that a proper GUI will enhance the tool’s usability. Also, in terms of research, GUI will enable researchers to design experiments of greater volume and complexity, up to the point of enabling comparisons of algorithms in new ways that will become accessible thanks to the GUI. When greater sets of options are revealed (for instance, variants of each algorithm explicitly listed and possible to include in an experiment), we believe researchers would choose to conduct experiments in ways they would not have explored otherwise.

In creating the new GUI for our tool, we aimed at creating a neat, modern look and feel. Typically for Computer Scientists, the most common type of GUI applications are Integrated Development Environments (IDEs). We have borrowed many elements from modern IDEs design. After going through several prototypes for the GUI, we feel like we have come upon an elegant, convenient and usable GUI design that would be comfortable to use and easy to learn.

When designing the GUI system we had two concerns in mind. First, to separate the development of the UI from that of the framework’s domain layer. We have used different tools to write code and to design GUI, and wanted it to be comfortable to combine domain code and GUI code. In order to achieve this we have made the set of dependencies (aka Java imports) for a GUI element as little as possible. As a result, the GUI can be developed independently without having to import the Framework’s JARs. A second concern was extensibility for future modules and plug-ins. This was achieved by modeling the GUI in a Server-Client pattern. Messages passed to and from the GUI are in the form of hashed dictionaries with Strings as keys and Objects as values. This requires the GUI and the Controller to agree upon a certain protocol when adding a new GUI element and does not limit the ways that the GUI may communicate with the controller. Below is the GUI code used for sending a message to the domain layer, which knows how to unpack the data out of it by identifying it as a "RunSimulation" message.
Fig. 1. A screen shot of the Graphical User Interface. On the left pane we can see a tree-representation of Problem Sets, Algorithms to solve them and statistics to measure during the execution, each with its own parameters. The tool is in a state of executing one of the problems, and is displaying the Network Graph visualization.

```java
private void buttonRunActionPerformed(java.awt.event.ActionEvent evt) {
    HashMap<String, Object> data = new HashMap<String, Object>();
    data.put("problemType", "CSP");
    data.put("p1", Double.valueOf(this.sliderP1.getValue() * 0.01));
    data.put("p2", Double.valueOf(this.sliderP2.getValue() * 0.01));
    data.put("numberOfVariables", Integer.valueOf(this.textFieldNumberOfVariables.getText()));
    data.put("domainSize", Integer.valueOf(this.textFieldDomainSize.getText()));
    data.put("numberOfProblems", Integer.valueOf(this.textFieldNumberOfProblems.getText()));
    data.put("visualization", "histogram");
    data.put("visualization", "nirein3");
    controller.request("RunSimulation", data);
}
```

Finally, a tag system is under development, which will allow declaring variables in the algorithm’s code as user-controlled. Such variables will then get a GUI element generated for them whenever the algorithm is used in an experiment. This will allow complex algorithms to be fully represented in our GUI,
without requiring the implementor of such algorithms to write any GUI code to integrate with our system but only to collect the specific data items the algorithm requires.

4 Visualization of DCR algorithms

Visualization of CP algorithms has been done before [14, 1, 4, 15], with the emphasis usually given to Post-Mortem analysis. For our team, visualization came into play when we found ourselves illustrating certain behaviors observed in our algorithms in a graphical manner. For instance, a (normal) distribution of agents' activity (be it NCCCs or changes of assignments): in a distributed algorithm. It is not uncommon that while the first and last agents in the order of the algorithm's run display very little activity, the middle agents display a large amount of activity. In other words, most of the search process is performed in the middle of the search space. Backtracking and reassignments for the earlier nodes are rare, while the lower parts of the search space do not get fully explored due to failures (inconsistencies) in shallower levels. Drawing a graph of this activity, with the x-axis being the linearly-ordered agents and the y-access being a measurement of some type of activity, a bell-shape distribution can be easily observed (See figure 2). Naturally, this calls for implementing some dynamic-ordering over the variables, allowing the agents to explore different parts of the search space concurrently.

While this is a very modest example, we would like to further research this, hoping to discover that other visualizations can grant us further insights regarding our algorithms behavior. Of course, the motivation for this visual exploitation apart from deeper understanding of a given algorithm, lies in inventing new heuristics or even new algorithms.

While such visual analysis for DCR execution data existed, we wanted to implement a dynamic visualization. We drew inspiration from the world of communication, where constellation diagrams are used for analyzing physical layer decoders performance. We expect real-time visualization of DCR search algorithms to provide an interesting view as well. Furthermore, post-mortem analysis relies on a post-execution dataset. Since execution can last from hours to days, we felt that an on-line, real-time visual display of an algorithm's behavior could be more beneficial to have. While this approach might result with visualizing incomplete data (as the on-screen display update rate cannot be as rapid as that of the underlying computation by the algorithms), we are aware that some data is lost and do not consider this loss a meaningful one. Our guiding line is not to have the visualization become an overkill for the system, straining it and thus affecting the algorithm's 'natural' behavior. We intend to extend the visualization system to allow recordings of executions and Post-Mortem analysis of data as well; but the heart and main use-case of this system lies in real-time visualization, side by side with the algorithms' execution.

All visualization modules comply with a given interface. The system architecture for visualization relies upon the statistics-gathering system. The visualiza-
tion module gets data pushed onto it by a polling thread (called the Visualization Driver) which constantly queries the statistics gathering system. Thus thread-safety and handling is guaranteed by the system, and is of no concern to the implementor of a new visualization module. A visualization module may be developed separately from the rest of the framework, feeding from a mockup data generator. Yet, it is still possible to plug into the actual framework.

Usage of external libraries is also available. For instance, one of our modules makes use of the JUNG (Java Universal Network/Graph) Framework. Examples of Visualizations:

– Histograms (Figure 2) updating in real time, a set of histograms can be displayed alongside executions, illustrating how each agent (or a subset of the agents) performs in a certain aspect. The measured parameter could be NCCCs, Assignments, Messages Sent or any other statistics measurement used in the execution.

Fig. 2. A screen shot of the Histogram visualization pane for an execution of the AFC algorithm. Each column represents the count of NCCCs performed by its corresponding agent. Visualization of this algorithm reveals an emerging pattern. The counts of the agents’ NCCCs quickly become relatively fixed, meaning that the search efforts are not evenly distributed between the agents, and that some agents will perform harder than the others during most of the algorithm’s execution.

– Network Graphs (Figure 3) a graph representing the algorithm’s behavior. Nodes represent a subset (possibly complete) of the agents set. Edges represent two agents which have exchanged messages. Nodes change their color and size according to the amount of the activity that they performed (NCCCs, assignments). Edges can appear and expand in width according to how many messages have been recently transferred between the two connected
agents. Figure 3 presents the Network Graph for a run of the AFC algorithm with 8 agents, each agent with domain size of 12, with p1,p2 values at 0.5. Messages sent to an agent trigger its message-handling; likewise, messages sent from an agent result from its activity. Thus, message exchange to and from an agent are evidences of the agent’s activity. In the problem being solved in the visualized execution, the constraints graph is not dense, allowing the first 3 agents to reach stable assignments early in the search process. This leaves the last 5 agents to perform most of the search.

Fig. 3. A screen shot of the Network Graphs visualization pane for an execution of the AFC algorithm. Each node represents an agent (node colors have been set to white). Each bold edge signifies that the two connected nodes (agents) have exchanged messages. The wider the edge, the more messages exchanged between the agents. Message exchange is an evidence of agents activity; thus, it can be observed in this figure that only 5 out of 8 agents are actively performing search, while agents 0, 1, 2 are passive.

Other visualization modules also exist and can be reviewed in our user manual.

5 Future Plans

- Networking and Distribution. The tool is currently designed to execute on a single machine. Extending it for working on a distributed group of computers is interesting for a number of reasons. A single machine might not be always able to handle an experiment (for instance, when the number of agents is large). Distribution of agents between a number of machines, with the agents
communicating over the network, is thus essential. Adding networking capabilities to the tool may allow a researcher to control execution of experiments on a remote machine (or better yet, a cluster). Having a central computer initializing simulations in a number of machines can also make it easier to manage sets of experiments that are distributed among several machines in order and are run in parallel to save time.

– Post-Mortem Analysis. As we mentioned, Post-Mortem analysis has been a major part of past efforts in visualizing DCR algorithms. Our current architecture allows us to extend the capabilities from real-time visualization to a discrete, controlled display of execution results from structured files that will be generated by our framework. This animation-like approach to visualization can be useful for educational purposes as well as for sharing results or ideas by exchanging these files. It is said that a single image is worth a thousands words; we aim to allow the achieving of these images easily.

– Web Interface. To truly make our tool useful for the general public, we would like to implement a web interface for it. Such an interface will make it more accessible to students and researchers alike, and would also allow those less technology-savvy to experiment with the tool. As many people from the multiple-agents community are now beginning to take interest in the field of DCR, we would like to have our tool ready to accept and serve those who can benefit from it.

– Debugger. In studying the behavior of DCR algorithms, we are often interested in seeing how agents react to changes and deviations from the standard scenario. Examples for this attitude include different message-passing policies executed by the mailer and heuristics such as dynamic ordering of the agents. A debugger module for our framework will consist of allowing to put breakpoints in an algorithm’s code such that at certain points an agent’s state can be both observed and manipulated. This will enable the programmer to empirically witness how state and state-changes affect an agent’s behavior and an algorithm’s overall behavior.
A Complete Implementation of the SBT Agent

```java
public class SBT_Agent extends AbstractAgent {

protected void process() {
    AbstractMessage message_from_mailbox = getMessageFromMailbox();
    if (message_from_mailbox instanceof SimulationFinished) {
        setFinished(true);
    } else { // Ordinary message for the algorithm to handle
        SBT_Message sbt_message = (SBT_Message) message_from_mailbox;
        ltc_ = sbt_message.getLTC() + 1;
        CPA cpa = sbt_message.getCPA();
        AbstractMessage message_to_send;
        if (sbt_message.getSender() > id) { // Backtrack Message
            current_assignment++;
            CPA cpa = sbt_message.getCPA();
            current_assignment = assign(cpa);
            AbstractMessage message_to_send = backtrack(cpa);
        } else { // Continue to the next agent
            message_to_send = nextAgent(cpa);
            sendMessage(message_to_send);
        }
    }
}

/** Trying to assign a value to this variable */
protected int assign(CPA cpa) {
    CSPProblem problem = (CSPProblem)m_problem;
    boolean consistent = false;
    int current_assignment = this.current_assignment;
    while (current_assignment < problem.getDomainSize(id) && !consistent) {
        consistent = true;
        for (int current_variable = 0; current_variable < id && !consistent;
             current_variable++) {
            accumulateAgentStatistics("NCCCs"); // Incrementing agent's statistics count of NCCCs.
            if (problem.isConstrained(id, current_variable)) {
                consistent = !(problem.isConflicted(id, current_assignment,
                        current_assignment, cpa.getAssignment(← current_variable)));
            }
        }
        if (!consistent) current_assignment++;
    }
    return current_assignment;
}

/** Backtrack to the previous agent */
protected AbstractMessage backtrack(CPA cpa) {
    cpa.removePartialAssignment(id);
    return new SBT_Message(id, id - 1, ltc, cpa);
}

/** Advance to the next agent */
protected AbstractMessage nextAgent(CPA cpa) {
    cpa.setPartialAssignment(id, currentAssignment);
    accumulateAgentStatistics("Assignments");
    return new SBT_Message(id, id + 1, ltc, cpa);
}
```

References

A Distributed Cooperative Approach for Optimizing a Network Game *

Alon Grubshtein and Amnon Meisels
Ben-Gurion University of the Negev,
Department of Computer Science,
P.O.B 653, Be'er Sheva, 84105, Israel
alongrub@cs.bgu.ac.il, am@cs.bgu.ac.il

Abstract. The present study considers a distributed cooperative approach for network problems where agents have personal preferences over outcomes. Such problems can be described by Asymmetric Constraints where the joint action of agents yields different gains to each participant [6]. The proposed method constructs and solves an Asymmetric Distributed Constraints Optimization Problem whose solutions guarantee a minimal gain for each agent, which is at least as high as the agents’ Bayesian equilibrium gain. The paper focuses on a special class of Network Games and proves that the proposed method produces optimal results in terms of the number of agents whose gain improves over their equilibrium gain and that the resulting solutions are Pareto Efficient. Extensive empirical evaluation of the studied network problem shows that the number of improving agents is not negligible and that under some configurations up to 70% of the agents improve their gain while none of the agents receive a payoff lower than their equilibrium gain.

1 Introduction

A key attribute of any Multi Agent System (MAS) is the level of collaboration agents are expected to adopt. When agents share a common goal it is natural for them to follow a fully cooperative protocol. A common goal can be the election of a leader, finding shortest routing paths, or searching for a globally optimal solution to a combinatorial problem (cf. [13]). When the agents have different and conflicting goals, competition and self interest are expected.

Distributed Constraints Satisfaction or Optimization Problems (DCSPs or DCOPs) are important examples of the cooperative model, where a set of agents attempt to satisfy or optimize a common global objective [11, 2, 6]. The Asymmetric Distributed Constrains Optimization Problems (ADCOPs) model provides a generalization of DCOPs where constrained agents gain differently (asymmetric gains) [6]. Such asymmetric constraint networks can be thought of as

* This research was supported by the Lynn and William Frankel Center for Computer Science, and by the Paul Ivanier Center for Robotics.
“cooperative network games”. In the context of a DCOP, it is natural to expect cooperative behavior when agents are interested in a global objective (e.g. the value of the objective) and to expect selfish or competitive behavior when agents are interested only in their personal gains.

An example of a competitive (or selfish) scenario are Network Games [8, 4] that assume that participants are only willing to take actions which improve (or do not worsen) their gains. In these situations it is customary to find an equilibrium state from which no agents would care to deviate [17]. These equilibria states are not necessarily efficient with respect to the global objective [18, 17] or even with respect to the personal gains of the participants (e.g. “the prisoners’ dilemma”) [15].

The present study focuses on a cooperative search approach for a class of network games [4]. The proposed approach is based on the construction of an ADCOP whose cooperative solution guarantees a minimal gain to each agent. To accommodate the natural selfish behavior of the participating agents, the proposed cooperative solution guarantees a personal gain for every participant the is not lower than the agent’s game theoretic equilibrium gain. That is, the proposed approach uses the standard assumptions on cooperation of ADCOP (and DCOP) agents, but is capable of bounding the personal gain of its solution according to a game theoretic result. Thus, this approach extends the common approach taken when cooperatively solving MAS problems [12, 2, 14, 19]. It applies the same set of assumptions needed to cooperatively solve a multi agent system but provides a different, user oriented, solution concept. It is important to note that the proposed ADCOP solution benefits all agents simultaneously - gains are at least as high and often higher than the equilibrium gains.

The proposed method is applied to a class of Network Game defined in section 2. Network Games have seen growing attention in recent years due to their ability to capture many real world problems such as File sharing, ad hoc P2P networks and network formation games [3, 10, 17].

The present study is to our best knowledge, a first attempt to use a distributed search method that guarantees an improvement in the personal gain of agents over their equilibrium (game theoretic) gain. It uses the definition of the Cost of Cooperation [5], which defines a measure of individual loss from cooperation. An ADCOP representation of the given network game is presented along with a proof that in its optimal solutions this cost is non positive (i.e. no agent stands to lose). We further prove that no other solution improves the gain of a greater number of agents. Empirical evaluation on a set of randomly generated network problems demonstrates that the fraction of improving agents can be quite large (up to 70%).

2 Download / Free Ride network interaction

Consider a set of agents wishing to download large amounts of information from a remote location. Agents may receive the information from other agents in their local neighborhood (if these neighbors have it) or directly download it
from a central hub. The process of downloading data from the hub requires significant bandwidth and degrades performance (resource consumption, battery, etc). In contrast, sharing information with peers does not result in any significant impact on the agents. Information exchange between agents is limited to directly connected peers only (no transfer of information to second degree neighbors).

The entire interaction can be captured by a graph \( G = (V, E) \) where each edge \( e_{ij} \) specifies a neighborhood relationship and a vertex \( i \in V \) represents an agent. \( x_i \) denotes the action (e.g., assignment) made by agent \( i \), and \( x_N(i) \) is the joint action taken by all of \( i \)'s neighbors. \(^1\) The degree of node \( i \) will be denoted by \( d_i \).

The network itself (the graph \( G = (V, E) \)) is not known to all agents. The total number of participants \( n = |V| \) is known and so is the (fixed) probability \( p \) for an edge to exist between any two vertices. Such graphs \( G = (n, p) \) are known as Poisson Random Graphs or an Erdős - Rényi network [8].

The gain from receiving information is taken to be unity and the cost of downloading it is \( c \) (\( c < 1 \)). Agents can either download the relevant information from the hub (assign action \( D \)) or wait for one of their neighbors to download it (assign action \( F \)). The payoffs are a function of the agent’s action and of its peers:

\[
  u_i(x_i, x_N(i)) = \begin{cases} 
    1 - c & \text{if } x_i = D \\
    1 & \exists j \in x_N(i) \text{ s.t. } x_j = D \\
    0 & \text{otherwise}
  \end{cases}
\]

That is, if the agent exerts effort and downloads the information its gain is \( 1 - c \). If, on the other hand, it does not download but one of its peers does, its gain is 1. Finally, if neither it nor any of its peers download the information its gain is 0.

This interaction models the local provision of information (or a local public good) where the agents’ actions are termed “strategic substitutes”. It is given as an example by Galeoti et al. [4] who recently studied the effect of the network on behavior and payoff in network games.

2.1 Solving the D/F interaction

Given the set of agents in the D/F interaction, what should the joint assignment be? While there are numerous possible responses which are naturally affected by the agents’ cooperation level, we first consider the two most important ones representing two opposing extremes.

**Fully cooperative agents** - In the fully cooperative setting, individual gains of participants are often used to compute the value of some social, or global welfare function. Two extreme examples of Social Welfare Functions (SWFs)

\(^1\) We specifically refrain from the common \( x_{-i} \) notation to represent a joint action by all players but \( i \), to emphasize that player \( i \) is only affected by the set of its neighbors.
are the “Utilitarian” and “Egalitarian” SWF [16]. Mathematically speaking, the utilitarian approach seeks to maximize the sum of individual utilities, while the egalitarian approach seeks to equalize individual utilities.

The fully cooperative approach was successfully applied to numerous multiagent problems. By casting the problem to a distributed constraint problem – defining agents, variables, domains and constraints – one can invoke any of the DCR algorithms and find the optimal (or near optimal) solution to the problem at hand.

In the D/F game, the optimal cooperative solution, maximizing the sum of all participants’ gains (or the maximal average gain of each one), is the Minimal Dominating Set (MDS). A dominating set of a graph $G = (V, E)$ is a set of nodes $S \subseteq V$ such that every node is either in $S$, or else a neighbor of at least one element of $S$. Finding such a set in a distributed fashion can be readily achieved by formulating the problem as a Distributed Constraint Optimization Problem (DCOP) in which constraints incur an extreme cost to joint assignments where an agent does not belong to the set $S$ or is a neighbor to one that is.

![Fig. 1. An example graph with $n = 8$ and $p = 0.45$](image)

The MDS of the graph $G = (8, 0.45)$ illustrated in Figure 1 includes only two vertices. In one such optimal solution $a_3$ and $a_6$ are members of the dominating set and when $c = 0.7$, their gain is 0.3, the gain of all other participants is 1 and the aggregated social gain is 6.6. However, despite the high average gain of 0.825, one may argue that this solution is inadequate for agents with personal utility functions and that the lower gains of $a_3$ and $a_6$ are in contrast to their relative importance. In such cases one may consider a different solution concept.

**Fully competitive agents** - The common approach taken when self interested entities are present, is to invoke Game Theory [17]. By accepting the fundamental axioms of game theory one is able to predict possible outcomes of an interaction. These outcomes are standardly assumed to be equilibrium points (in terms of agents’ actions) from which no agent is willing to deviate. The best known solution concept is the Nash Equilibrium (NE) point, but there are many other equilibrium points which fit different aspects of the interaction (cf. [17]).

In a recent work by Galeotti et. al [4] the D/F interaction is analyzed and a unique Bayesian Nash Equilibrium (BNE) is computed. Based on the global
probability $p$ for an edge to exist between two neighbors, each agent calculates the probability for a randomly selected neighbor to be of degree $k$:

$$Q(k; p) = \binom{n-1}{k-1} p^{k-1} (1 - p)^{n-k-1}$$

Using this information Galeotti et. al define a parameter $t$ which is the smallest integer such that:

$$1 - \left[ 1 - \sum_{k=1}^{t} Q(k; p) \right]^t \geq 1 - c$$

In the unique BNE for the D/F network game, any participant of degree $k \leq t$ selects strategy $D$ and any participant of degree $k > t$ selects $F$.

When applied to the interaction instance illustrated in Figure 1 (cost value of $c = 0.7$) one first calculates the threshold value $t = 2$ from the above equations. This value defines the assignment for each agent in the BNE solution: $a_1$ and $a_8$ will assign action $D$ while all others will assign $F$.

Agents in this setting maximize expected gain which may result in low actual gain. In the BNE solution of the example problem agents $a_2, a_4, a_5$ and $a_7$’s gain is actually 0, while both $a_3$ and $a_6$ actual gains are higher than their expected gains. The aggregated social gain in the game theoretic approach is 2.6 with an average gain of 0.325. Although it is easy to see that one can cooperatively improve this BNE solution, reverting to the MDS solution will increase the overall sum of gains (average gain of agents) at the expense of $a_3$ and $a_6$. That is, it will penalize $a_3$ and $a_6$ and reduce their gains when acting with no cooperating protocol.

3 A cooperative framework for agents with personal utility functions

The cooperative and competitive approaches described in the previous section suffer from several problems. The cooperative approach blindly optimizes a pre-defined objective function with little or no care for the end cost of agents. This leads to a situation where results may be biased towards or against some agents who are assigned extreme costs. On the other hand, game theoretic solutions are often inefficient as in our example. There are numerous authors who claim that game theoretic solutions fail to faithfully represent reality or predict strategies of real world players [1, 7].

The investigation of the inefficiency of an equilibrium was triggered by Koutsoupias and Papadimitriou [9] who later coined the term “Price of Anarchy” (PoA). The PoA is a widely accepted measure for the inefficiency of an equilibrium with respect to some global objective function [18, 17]. It is computed as the ratio between the worst possible NE and the social optimum (e.g., the ratio between the sum of gains in the worst NE and the sum of gains in the socially optimal utilitarian gain). While this provides an informative measure on the social outcome it provides little insight into the inefficiency of an equilibrium from the point of view of the game participants.
The present paper focuses on the latter point. More specifically, we follow the ideas presented in [5] which define the Cost of Cooperation for agents:

**Definition 1 (Cost of Cooperation (CoC)).** Given a global objective function \( f(x) \), the Cost of Cooperation of an agent is defined as the difference between the lowest gain that the agent can receive in any equilibrium of the underlying game (if any, otherwise zero) and the lowest gain this agent receives from a cooperative protocol’s solution \( x \) that maximizes \( f(x) \).

The CoC provides a measure of the agents’ gain from cooperation. In the present paper, this measure is used to direct a cooperative search process by examining the expected actions and gains of self interested rational users and disallowing solutions which result in lower gains than those of the worst stable game – for any of the agents. One can view cases that may encourage cooperation as problems that have solutions with non positive CoC.

The proposed method, constructs and solves an ADCOP for each D/F network game guaranteeing non positive CoC for all agents on the network. An ADCOP is an extension to the DCOP paradigm which addresses individual agent’s gains and captures game-like interactions between agents [6].

By cooperatively solving the following ADCOP non positive CoC is guaranteed to all agents:

- A set of Agents \( A = \{a_1, a_2, ..., a_n\} \) - each holds a single variable and corresponds to a single user.
- A set of Domains \( D = \{D_1, D_2, ..., D_n\} \) for the variables held by the agents. Each domain consists of only two possible values: \( \{D, F\} \).
- A set of asymmetric constraints \( C \). Each constraint \( c_i \) is identified with a specific agent (e.g. agent \( i \)) and defined as a constraint of arity \( (d_i + 1) \) over all of agent \( i \)'s neighbors. The set of agents involved in constraint \( c_i \) is \( A_{c_i} = a_i \cup x_N(i) \). That is, the ADCOP has \( n \) constraints and each agent contributes to \( d_i + 1 \) different constraints. The costs associated with each constraint that add up to the total cost of \( c_i \) for agent \( i \) are summarized in the table in Figure 2.

<table>
<thead>
<tr>
<th>Costs of ( c_i )</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n^3 )</td>
<td>( \forall a_j \in A_{c_i}, x_j = F ) (all agents of the constraint assign ( F ))</td>
</tr>
<tr>
<td>( d_j )</td>
<td>\textbf{for each} ( a_j \in A_{c_i} ) with ( d_j &gt; t ) and ( \forall a_k \in x_N(j), d_k &gt; t ) that assign ( x_j = D )</td>
</tr>
<tr>
<td>( (n-1)n )</td>
<td>\textbf{for each} ( a_j \in A_{c_i} ) with ( d_j &gt; t ) and ( \exists a_k \in x_N(j), d_k \leq t ) that assign ( x_j = D )</td>
</tr>
<tr>
<td>1</td>
<td>\textbf{for each} ( a_j \in A_{c_i} ) with ( d_j \leq t ) which assigns ( x_j = D )</td>
</tr>
</tbody>
</table>

**Fig. 2.** Constraint’s costs for each combination of assignments
Note: costs of the ADCOP’s solutions (full or partial) are completely unrelated to the agents’ gains.

Let us now proceed to prove that finding a minimal assignment (an optimal solution) for the above ADCOP results in gains which are at least as high as the actual gains resulting from applying the unique BNE strategies for all agents.

Let the optimal solution to the ADCOP be denoted by $x$. The term null neighborhood defines a situation in which an agent and all of its connected peers assign $F$ as their action.

**Lemma 1 (No null neighborhoods).** In the optimal solution to the ADCOP there exists at least one agent in the local neighborhood of each agent which assigns $D$ (specifically, the agent itself can assign $D$).

**Proof.** Assume by negation that there exists an agent $a_i$ with a null neighborhood in $x$. This means that the cost of $c_i$ (the constraint originating from $a_i$) is $n^3$. Let $x'$ be a complete assignment which differs from $x$ in the assignment of an agent $a_j \in A_{c_i}$. That is, $x'_j = D$. As a result the previously null neighborhood of $a_i$ in $x$ is not part of $x'$.

One can compute an upper bound on the cost of $x'$ by assigning this change to an agent $a_j$ with $d_j = n - 1$ (where $n - 1 > t$) which has at least one neighbor $a_k \in x_N(j)$ of degree $d_k \leq t$ (i.e. $a_j$ appears in all $n$ constraints of the problem). The resulting cost of $x'$:

\[
\text{COST}(x') = \text{COST}(x) - n^3 + (n - 1)n^2
\]

which implies that $\text{COST}(x') < \text{COST}(x)$ in contradiction to the optimality of $x$.

An important implication of Lemma 1 is that in the optimal solution to the ADCOP no agent receives a payoff of 0. That is, the ADCOP’s optimal solution results in a dominating set.

**Lemma 2.** The gain of agents $a_i$ with degree $d_i \leq t$ in $x$ is at least as high as its actual gain in the BNE.

**Proof.** In the BNE $a_i$’s gain is exactly $1 - c$ ($d_i \leq t$ and hence $a_i$ assigns $D$). The only possible lower gain is 0. However, this is the gain of an agent with a null neighborhood and hence (following Lemma 1) its gain must be at least $1 - c$ when cooperating (either it will assign $D$ or one of its peers will).

**Lemma 3.** The gain of agent $a_i$ with degree $d_i > t$ in $x$ is at least as high as its actual gain in the BNE.

**Proof.** Due to its degree, $a_i$’s BNE gain is never $1 - c$. When all its neighbors $a_j \in x_N(i)$ are of degree $d_j > t$ its BNE gain is 0, and when at least one neighbor is of degree $d_j \leq t$ its BNE gain is 1. Since the ADCOP gain is always higher than 0 (Lemma 1) we only consider the case when $a_i$’s gain is 1 in the BNE.
This is only attainable when \( a_i \) has at least one neighbor \( a_j \) with degree \( d_j \leq t \). Thus, to conclude our proof we have to show that \( a_i \)'s ADCOP gain is not \( 1 - c \).

By the problem description this gain can only occur when \( a_i \) assigns \( D \) in \( x \). This assignment incurs a positive contribution on the ADCOP only when it prevents the existence of a null neighborhood for \( a_i \), or when it prevents it for one (or more) of \( a_i \)'s neighbors.

If \( a_i \) has a null neighborhood which it would like to prevent, there must be a neighbor \( a_j \in x_N(i) \) (with degree \( d_j \leq t \)) which assigns \( F \) in the ADCOP. In this case we define a new solution \( x' \), which differs from \( x \) in the assignments of \( a_i \) and \( a_j \) (i.e., \( x'_i = F \) and \( x'_j = D \)). \( x' \)'s cost is:

\[
COST(x') = COST(x) - (d_i + 1)(n - 1)n + d_j
\]

The highest cost \( x' \) can take will be when \( d_i = 1 \) and \( d_j = n - 1 \) in which case:

\[
COST(x') = COST(x) - 2(n - 1)n + (n - 1) = COST(x) - (n - 1)(2n - 1)
\]

implying that \( a_i \) does not assign \( D \) to prevent a local null neighborhood in the optimal solution.

When \( a_i \) assigns \( D \) to prevent the existence of a null neighborhood for one of its neighbors, we classify the neighbor's type:

1. The neighbor \( a_j \) has at least one neighbor \( a_k \) with degree \( d_k \leq t \). We generate the assignment \( x' \) in which \( x_i = F \) and \( x_k = D \). As before the cost of \( x' \) is lower than that of \( x \) since \( a_k \) contributes a single unit per each constraint it is involved in (at most \( n \) units) whereas the assignment change of \( a_i \) lowers the cost by at least \( (n - 1)n \) units in contradiction to the optimality of \( x \).

2. The neighbor \( a_j \) has no neighbor \( a_k \) with degree \( d_k \leq t \). Assigning \( x_j = D \) and \( x_i = F \) to \( a_i \) lowers the resulting cost since the contribution of \( a_j \) to the cost is bounded by \( n^2 \) (it incurs a cost of \( d_j \) to each of the \( d_j + 1 \) constraints it is involved in), whereas the gain from changing the assignment of \( a_i \) is \( (d_i + 1)(n - 1)n \).

This means that even if all neighbors are of the second type (which always incur greater costs), and they are all connected to all other agents, the cost of the modified solution is:

\[
COST(x') = COST(x) - (d_i + 1)(n - 1)n + d_i \cdot n(n - 1)
\]

Hence we conclude that that the gain of \( a_i \) will not be reduced from \( 1 \) in the BNE to \( 1 - c \) (or \( 0 \)) in the ADCOP’s optimal solution.

**Theorem 1 (Gain guarantee).** In the optimal solution to the ADCOP described above for the D/F interaction, all agent’s CoC is non positive with respect to the actual gains in the BNE.

**Proof.** Since the BNE is unique, it also provides the lowest gain that each agent can receive in an equilibrium. Thus, by directly following Lemmas 1-3 our claim is proved.
Figure 3 presents a randomly generated example of a D/F network of 50 participants with an average of 5 neighbors per participant $G=(50,0.1)$. The cost of exerting effort (assign action $D$) is $c=0.7$. The example in Figure 3 demonstrates the differences in gains with respect to the BNE in two alternative solutions. One alternative applies and solves the proposed ADCOP. The other alternative is the socially optimal MDS solution. The Gray nodes represent agents which received higher gains than their BNE gain, Black nodes represent agents which did not improve their gains and finally White nodes represent lower gains than those attained in the BNE. Two important features are illustrated by this example:

1. There is a significant number of agents (nearly 40%) improving their gains in the ADCOP solution (Figure 3(a)) while none attain lower gains.
2. The socially (cooperative) optimal solution (Figure 3(b)) increases the gains of even more participants, however, it also reduces the gains of nearly 20% of the agents.

Having proved that the gains of agents can only improve, the following two theorems address the question of efficiency of the proposed scheme:

**Theorem 2 (Pareto efficiency).** The results of the proposed method are Pareto efficient. That is, there is no solution which further increases the gain of some agents without reducing the gain of at least a single agent.

**Proof.** To prove our claim, we need only consider agents with an ADCOP gain of $1-c$. A gain of 1 cannot be further improved, and following Lemma 1 there are no agents with 0 gain in the ADCOP solution.
Let $x$ be an optimal solution to the ADCOP and assume by negation that there exists a solution $x'$ in which at least one agent gains more and none gain less. By our previous observation, the improving agents assign $F$ in $x'$.

According to Lemma 1 there can be no null neighborhoods, and thus we must assume that the change in assignment of the improving agents does not result in one. This, in turn, implies that $\text{COST}(x) < \text{COST}(x')$ since the improving agents no longer contribute to the ADCOP cost of the solution (Figure 2) - in contradiction to the optimality of $x$.

**Theorem 3 (Efficiency).** The proposed solution is optimal in terms of the number of agents that improve their gain. That is, there is no other solution in which a larger number of agents have negative CoC.

*Proof.* Again we note that agents with a BNE gain of 1 have a maximal gain and that all agents with 0 BNE gain improve their gain (Lemma 1). We also note that retaining BNE null neighborhoods will not improve the gains of any agent (assigning $D$ in any null neighborhood can only increase gains) and that following our previous theorem there is a minimal number of agents with an ADCOP gain of $1 - c$. Thus, we need to prove that the ADCOP solution also improves the maximal number of agents with a BNE gain of $1 - c$. That is, we need only show that the maximal number of agents which assign $D$ in the BNE change their assignment to $F$.

We again assume by negation that there exists a solution $x'$ in which the number of agents changing their assignment to $F$ is greater than that of the ADCOP’s optimal solution $x$. If there was such a solution it would imply that the cost of $x'$ is lower than that of $x$, and hence would contradict the optimality of $x$. The existence of such a solution is therefore impossible.

### 4 Experimental evaluation

The proofs presented in the previous section demonstrate that the proposed approach is capable of finding a joint action improving the gain of all agents who can do so without introducing a negative impact on others. However, this guarantee does not answer the qualitative question: what is the actual number of agents improving their gain?

To empirically evaluate the effectiveness of the proposed approach, one can use a large set of randomly generated interactions. These interactions are characterized by the interaction graph $G = (n, p)$ and the cost $c$ associated with assigning the action $D$.

Random graphs with 30 agents ($n = 30$) were generated for a variety of density values ($p$). For each density value the cost value $c$ was varied and the joint BNE assignment was calculated. Results of each configuration were averaged over 50 instances and in each run the gain of agents was compared to the gain reached by the ADCOP solution.

Figure 4 presents the average number of agents with negative CoC (these represent all agents who could potentially improve their gain without incurring
costs on others). The results indicate that the proposed method indeed improves the gain of a significant number of agents and that this occurs for all values of density that were tried. As cost value increases the number of agents with increased gain dramatically increases. This is due to the rather moderate increase in the threshold values $t$ occurring when costs are higher (Figure 5). These values become significantly lower than the mean degree value ($E[d_i] = p \cdot n$) which in turn drives more agents to the assignment $F$ in the BNE. The resulting joint BNE action has therefore higher chances of including null neighborhoods which are not present in the ADCOP solution. Indeed our run logs show a significant number of agents improving their gains from 0 to 1 implying that under some parameters a large number of null neighborhoods exist.

The qualitative assessment of the change in the gain of agents can also be examined for the cooperative optimal solution - the Minimal Dominating Set (MDS). In this case, one must also account for the fact that some agents may lose from the MDS solution.

Figure 6 presents the average impact on the gain of agents for the same set of interactions when the joint action taken is the MDS solution. The trend on the number of agents with increased gains (Figure 6(a)) is somewhat similar to that of the ADCOP’s, although this number is consistently larger for the MDS. However, when one examines the average loss (calculated as the ratio between the number of agents with gains lower than that of the BNE to the number of agents with non zero gains in the BNE) it is evident that the MDS incurs a high cost on agents: it will almost always result in a gain reduction of 10% - 30% of the agents with non zero BNE gains.
5 Discussion

A new approach to distributed cooperative optimization is proposed where agents interact in a family of network games and have personal utility functions. The proposed approach guarantees a minimal gain equivalent to the equilibrium gain of each participant (non positive Cost of Cooperation) by forming and cooperatively solving an Asymmetric DCOP.

This guarantee is realized on a class of Network Games where agents gain by either free riding other agents’ efforts or download information for themselves. Although any equilibrium point defines a non positive CoC solution, our proposed ADCOP is proved to yield Pareto Efficient solutions which also maximize the number of agents with improved gains over their game theoretic (e.g. equilibrium) gains.

Experimental evaluation of the D/F interaction under different configurations demonstrates the effectiveness of the proposed approach. A large fraction (up to 70%) of the agents benefit from it while none gain less than their game theoretic gain. In contrast, the socially maximizing Minimal Dominating Set solutions benefit an even larger fraction of the agents but do so at the expense of up to 30% of the agents whose gains become lower.

Note that a greedy hill climbing search starting from the game theoretic equilibrium state may also result in a Pareto Efficient solution, however, it is not guaranteed to find the solution benefiting the maximal number of agents $^2$.

$^2$ Currently, there is no distributed hill climbing search algorithm for ADCOPs. Existing algorithms such as MGM or k-Opt lose their monotonicity property when asymmetric gains are involved [6]
Fig. 6. The average gains and losses of agents in the cooperatively optimal solution (Minimal DS).

The D/F interaction can be further enhanced to include improved gains when more agents download the information. The resulting interaction can provide a better approximation of P2P networks and their interactions. In such cases one needs to re-calculate the $t$ value again and describe a generalized form of the proposed ADCOP which achieves non positive CoC solutions.

References

On Message-Passing, MAP Estimation in Graphical Models and DCOPs

Akshat Kumar, William Yeoh, and Shlomo Zilberstein

Computer Science Department
University of Massachusetts
Amherst, MA 01003
{akshat,wyeh,shlomo}@cs.umass.edu

Abstract. The maximum a posteriori (MAP) estimation problem in graphical models is a problem common in many applications such as computer vision and bioinformatics. For example, they are used to identify the most likely orientation of proteins in protein design problems. As such, researchers in the machine learning community have developed a variety of approximate algorithms to solve them. On the other hand, distributed constraint optimization problems (DCOPs) are well-suited for modeling many multi-agent coordination problems such as the coordination of sensors in a network and the coordination of power plants. In this paper, we show that MAP estimation problems and DCOPs bear strong similarities and, as such, some approximate MAP algorithms such as iterative message passing algorithms can be easily tailored to solve DCOPs as well.

Keywords: Graphical Models, MAP Estimation Problems, DCOPs

1 Introduction

Markov random fields (MRFs) [32] is a graphical model, where nodes in a graph correspond to random variables and edges in a graph correspond to potential functions between pairs of random variables. A common problem in MRFs is to compute the maximum a posteriori (MAP) assignment, which is the most probable assignment to all the random variables of the underlying graph. The MAP estimation problem is a problem common in many applications such as computer vision, bioinformatics and information theory. For example, they are used to identify the most likely orientation of proteins in protein design problems [35; 25]. As such, researchers, who are typically from the machine learning community, have developed a variety of approximate inference algorithms [32; 24] to solve them.

Distributed constraint optimization problem (DCOP) [15; 18; 38] is also a graphical model; nodes in a graph correspond to agents, where each agent can take on a value, and edges in a graph correspond to constraint functions between pairs of agents, where each constraint function is a function of the values of a pair of agents. The agents in a DCOP coordinate with each other via messages
to optimize all their constraint functions. DCOPs have emerged as a popular model for representing multi-agent coordination problems where the primary interactions are between subsets of agents. Examples of such problems include the scheduling of meetings [13], the coordination of sensors in networks [3], the management of power plants [11] and the generation of coalition structures [27]. As such, researchers, who are typically from the multi-agent systems community, have developed a variety of optimal DCOP algorithms [14; 15; 18] and approximate DCOP algorithms [4; 6; 16] to solve them.

Since both MAP estimation problems and DCOPs are graphical models, they share many similarities. The main difference is MAP estimation problems are centralized problems while DCOPs are decentralized problems. However, many (centralized) algorithms used to solve MAP estimation problems can be executed in a decentralized fashion and they can thus be tailored to solve DCOPs. For example, the max-sum algorithm [3] and algorithms using the Divide-and-Coordinate approach [28] that are used to solve DCOPs are motivated by MAP estimation algorithms [17; 10]. In this paper, we aim to highlight the similarities between the MAP estimation problem and DCOPs more explicitly, and show how a class of approximate MAP algorithms, namely iterative message passing algorithms, can be tailored to solve DCOPs. To the best of our knowledge, the connection between MAP estimation problems and DCOPs have not been explicitly made. Thus, it is our hope that this work will better bridge the two research communities, namely machine learning and multi-agent systems, and will help cross-fertilize them.

This paper is organized as follows. In Section 2, we describe MAP estimation problems, DCOPs and their similarities. In Sections 3 and 4, we give a brief overview of several iterative message passing algorithms and how they can be applied to solve DCOPs, and in Section 5, we describe their properties and space complexities. Lastly, we present our experimental results and conclusions.

2 Graphical Models

Probabilistic graphical models provide an effective framework for compactly representing probability distributions over high dimensional spaces and performing complex inference using simple local update procedures. In this work, we relate two optimization problems represented as graphical models: the maximum a posteriori (MAP) estimation in Markov random fields (MRFs) [32] and distributed constraint optimization problems (DCOPs) [15; 18; 38]. MAP estimation is crucial for many practical applications in computer vision and bioinformatics such as protein design [35; 25]. Computing the MAP exactly is NP-hard for general graphs [2]. Thus, approximate inference algorithms are often used [32; 24]. In this section, we will provide an overview of MAP estimation in MRFs and how they relate to DCOPs.
2.1 MRFs and MAP Estimation Problems

A pairwise Markov random field (MRF) can be visualized by an undirected graph \( G = (V, E) \). It is formally defined by

- A set of random variables \( X = \{x_i \mid \forall i \in V\} \), where each random variable has a finite domain of possible values that it can be assigned. Each random variable \( x_i \) is associated with node \( i \in V \).
- A set of potential functions \( \theta = \{\theta_{ij}(x_i, x_j) \mid \forall (i, j) \in E\} \). Each potential function \( \theta_{ij}(x_i, x_j) \) is associated with edge \((i, j) \in E\).

The complete assignment \( x \) to all the random variables has the probability:

\[
p(x; \theta) \propto \exp \left( \sum_{ij \in E} \theta_{ij}(x_i, x_j) \right) \tag{1}
\]

The objective of a maximum a posteriori (MAP) problem is to find the most probable assignment to all the variables under \( p(x; \theta) \). This objective is equivalent to finding a complete assignment \( x \) that maximizes the function:

\[
f(x; \theta) = \sum_{ij \in E} \theta_{ij}(x_i, x_j) \tag{2}
\]

Additionally, we assume without loss of generality that each \( \theta_{ij} \) is non-negative in this paper. Otherwise, a constant can be added to each \( \theta_{ij} \) without changing the optimal solution.

2.2 DCOPs

Like MRFs, a distributed constraint optimization problem (DCOP) with binary constraints can also be visualized by an undirected graph \( G = (V, E) \), commonly called a constraint graph. It is formally defined by

- A set of agents \( X = \{x_i \mid \forall i \in V\} \), where each agent has a finite domain of possible values that it can take on. Each agent \( x_i \) is associated with node \( i \in V \).
- A set of constraint functions \( \theta = \{\theta_{ij}(x_i, x_j) \mid \forall (i, j) \in E\} \). Each constraint \( \theta_{ij}(x_i, x_j) \) is associated with edge \((i, j) \in E\).\(^1\)

Therefore, agents and constraint functions in a DCOP correspond to random variables and potential functions in an MRF, respectively. Similar to the MAP estimation problem, the objective in a DCOP is to find the complete assignment

\(^1\) Although the typical notation of a constraint function is \( F_{ij} \) or \( c_{ij} \) in the DCOP literature, we use the notation in the machine learning literature to better illustrate the mapping between MAP estimation problems and DCOPs.
that maximizes the function \( f(x; \theta) \) in Equation (2). The main difference between MAP estimation problems and DCOPs is that the former are centralized problems while the latter are decentralized problems. In MAP estimation problems, a single agent has complete knowledge of all potential functions and controls the value assignments of all random variables, while in DCOPs, each agent has knowledge of the constraint functions that it is involved in only and chooses its own value only. Nonetheless, many (centralized) MAP estimation algorithms can be executed in a decentralized fashion and can thus be tailored to solve DCOPs.

3 Variational MAP Formulations and Algorithms

In this section, we provide an overview of two common variational formulation of the MAP estimation problem: linear programming (LP) and quadratic programming (QP) formulations. Most of the existing algorithms in the machine learning literature can be classified as solving either one of these two formulations. We then describe two such algorithms, namely the max-product linear programming (MPLP) algorithm of [5], which operates on the LP formulation, and the expectation-maximization (EM) algorithm of [12], which operates on the QP formulation.

3.1 Linear Programming Formulation

We now describe the first common variational formulation of the MAP estimation problem: a linear programming (LP) formulation. We first briefly describe the concept of marginal polytope that is often associated with the MAP estimation problem. The reader is referred to [32] for more details.

Let \( \mu \) denote a vector of marginal probabilities (also called mean parameters) for each node and edge of the MRF. That is, it includes \( \mu_i(x_i) \forall i \in V \) and \( \mu_{ij}(x_i, x_j) \forall (i, j) \in E \). The set of \( \mu \) that arises from some joint distribution \( p \) over all the variables of the MRF is referred to as the marginal polytope \( \mathcal{M}(G) \):

\[
\mathcal{M}(G) = \{ \mu \mid \exists p(x) \text{ s.t. } p(x_i, x_j) = \mu_{ij}(x_i, x_j), p(x_i) = \mu_i(x_i) \} \quad (3)
\]

The MAP estimation problem is then equivalent to solving the following LP:

\[
\max_{x} f(x; \theta) = \max_{\mu \in \mathcal{M}(G)} \mu \cdot \theta = \max_{\mu \in \mathcal{M}(G)} \sum_{ij \in E} \sum_{x_i, x_j} \mu_{ij}(x_i, x_j) \theta_{ij}(x_i, x_j) \quad (4)
\]

Notice that \( p \) is a joint distribution over all the variables of the MRF – \( p(x_1, \ldots, x_n) \) – and is, in general, very hard to represent and reason with. It can be shown that there always exists a maximizing solution \( \mu \) for the above problem that is integral and gives the optimal assignment \( x \). Unfortunately, the number of constraints used to describe this polytope are exponential and, thus, it can
not be solved efficiently. To remedy this problem, researchers have proposed LP relaxations that outer bound the polytope \( \mathcal{M}(G) \). That is, the relaxed polytope \( \mathcal{M}_L(G) \) is a super set of \( \mathcal{M}(G) \) and may include certain fractional assignment to the random variables. This relaxation weakens the global constraint that \( \mu \) arises from some common distribution \( p \). Instead, only pairwise (corresponding to the edges) and singleton consistency is required for mean parameters as given by the following condition:

\[
\sum_{x_i} \mu_i(x_i) = 1 \quad \forall i \in V \quad (5)
\]

\[
\sum_{\hat{x}_i} \mu_{ij}(\hat{x}_i, x_j) = \mu_j(x_j) \quad \forall x_j, (i, j) \in E \quad (6)
\]

\[
\sum_{\hat{x}_j} \mu_{ij}(x_i, \hat{x}_j) = \mu_i(x_i) \quad \forall x_i, (i, j) \in E \quad (7)
\]

The constraints \( \sum_{\hat{x}_i} \mu_{ij}(\hat{x}_i, x_j) = \mu_j(x_j) \) and \( \sum_{\hat{x}_j} \mu_{ij}(x_i, \hat{x}_j) = \mu_i(x_i) \) ensure that, for each edge \((i, j) \in E\), the probability distribution \( \mu_{ij} \) of the edge is consistent with the probability distributions \( \mu_j \) of node \( j \in V \) and \( \mu_i \) of node \( i \in V \), respectively. It might appear we can easily solve the LP of Eq. (4) subject to the above constraints, which are polynomial in the number of edges and domain size. However, even for moderately sized graphs, this LP becomes quite large with many constraints and the black-box LP solvers such as CPLEX do not scale well [35]. Therefore such LP is solved using specialized message-passing algorithms that take into account the graph structure of this LP [21; 25; 23].

There are several other algorithms which either work on this LP formulation or its dual such as the tree-reweighted max-product (TRMP) algorithm [30], convergent tree-reweighted algorithm (TRW-S) [9], the max-sum diffusion algorithm (MSD) [33; 34] and Lagrangian relaxation based approaches [7] among others.

### 3.2 Quadratic Programming Formulation

We now describe the second variational formulation of the MAP estimation problem: a quadratic programming (QP) formulation. The reader is referred to [22] for more details. Instead of Equation (4), the MAP estimation problem can also be formulated as a QP:

\[
\max_{\mu_1, \ldots, \mu_n} \sum_{ij \in E} \sum_{x_i, x_j} \mu_i(x_i) \mu_j(x_j) \theta_{ij}(x_i, x_j) \quad (8)
\]

subject to \( \sum_{x_i} \mu_i(x_i) = 1, \mu_i(x_i) \geq 0 \ \forall i \in V \)

The above QP is compact even for large graphical models and has simple linear constraints – \( O(|V|k) \) variables, where \( k \) is the maximum domain size, and \( |V| \)
normalization constraints. Ravikumar and Lafferty also show that this formulation is exact [22]. That is, the global optimum of the above QP will maximize the function $f(x; \theta)$ and an integral MAP assignment can be extracted from it. However, this formulation is non-convex and thus makes the global optimization hard. Nonetheless, the local optima of this QP correspond to good solutions for most problems empirically [12].

Next we describe two algorithms, the max-product LP (MPLP) [5] and the expectation-maximization (EM) [12] which work on these two formulations. Both of these algorithms are based on iterative message-passing among neighboring agents and can be easily adapted for DCOPs.

### 3.3 MPLP and EM

Algorithms 1 and 2 show the pseudocode of the MPLP and EM algorithm, respectively. The MPLP algorithm operates on the dual of the LP formulation described in Section 3.1 and the EM algorithm operates on the QP formulation described in Section 3.2. In each iteration, every node $i$ sends a message to each of its neighbor $j \in \mathcal{N}(i)$. The messages in MPLP are denoted by $\gamma_{i \to j}$ and the messages in EM are denoted by $\delta_{i \to j}$. The marginal probability $\mu_i(x_i)$ in Algorithm 2 is the same as in the QP and the variable $C_i$ is the normalization constant such that the resulting marginal probabilities sum up to one. Once the
Algorithm 3: Max-Sum (MS) Algorithm

**input:** Graph $G = (V, E)$ and potential functions $\theta$

**output:** Return complete assignment $x$ s.t. $x_i = \text{argmax}_{\hat{x}_i} b_i(\hat{x}_i)$

repeat
  foreach node $i \in V$ do
    Send message $m_{i \rightarrow j}$ to each neighbor $j \in \text{Ne}(i)$:
    $m_{i \rightarrow j}(x_j) \leftarrow \max_{x_i} \left[ \theta_{ij}(x_i, x_j) + \sum_{k \in \text{Ne}(i) \setminus j} m_{k \rightarrow i}(x_i) \right]$
    Set node belief $b_i(x_i)$ to the sum of incoming messages:
    $b_i(x_i) \leftarrow \sum_{k \in \text{Ne}(i)} m_{k \rightarrow i}(x_i)$
  until desired number of iterations

algorithm is run for a desired number of iterations, the complete assignment $x$ can be extracted using the individual maximizer of the beliefs $b_i$ for MPLP and marginal probabilities $\mu_i$ for EM of each node $i$. Alternatively, one can also run a better scheme to extract the solution of EM by running an additional iteration of message passing among neighbors [22]. It is quite clear that these algorithms can be easily adapted to solve DCOPs by letting each agent control its corresponding random variable and manage the messages that it sends and receives. Furthermore, the messages are computed based only on the shared information among immediate neighbors in the graph. Therefore, this property follows the DCOP specification that each agent knows only about the constraint functions with its immediate neighbors.

4 Reparameterization View of the MAP Problem

Since the max-sum (MS) algorithm [17] has been very successful in several contexts such as information theory [1; 31] and DCOPs [3], we provide an overview of some of the recently developed theoretical results behind it. Unlike the MPLP and EM algorithms, max-sum reparameterizes the original probability distribution of Equation (1) in terms of max-marginals for acyclic graphs and pseudo-max-marginals for cyclic graphs before estimating the MAP from these (pseudo-)max-marginals.

Algorithm 3 shows the pseudocode of the MS algorithm. Like the MPLP and EM algorithms, in each iteration, every node $i$ sends a message $m_{i \rightarrow j}$ to each of its neighbor $j \in \text{Ne}(i)$. Once the algorithm is run for a desired number of iterations, the complete assignment $x$ can be extracted using the individual maximizer of the beliefs $b_i$ of each node $i$. Similar to the MPLP and EM algorithms, it is quite clear that this algorithm can also be easily adapted to solve DCOPs in the same way, as is already demonstrated [3].

We now briefly describe a reparameterization of the MAP estimation problem. The reader is referred to [31] for more details. Instead of max-sum, we discuss the max-product algorithm – max-sum messages can be considered as the log of messages in max-product which changes products to sums – which is more commonly described in the literature [31]. The max-product algorithm
maximizes the objective function of Equation (1) instead of the one of Equation (2). For comparison, their equivalent messages are:

\[
m^\text{sum}_{i \rightarrow j}(x_j) \leftarrow \max_{x_i} \left( \theta_{ij}(x_i, x_j) + \sum_{k \in Ne(i) \setminus j} m^{}_{k\rightarrow j}(x_i) \right)
\]

\[
m^\text{prod}_{i \rightarrow j}(x_j) \leftarrow \max_{x_i} \left( \exp(\theta_{ij}(x_i, x_j)) \prod_{k \in Ne(i) \setminus j} m^{}_{k\rightarrow j}(x_i) \right)
\]

where the former are the messages of max-sum and the latter are the messages of max-product. Assume that the algorithm converges and \( m^* \) are the messages upon convergence. We then define functions \( T^*_i \) for each node \( i \) and \( T^*_{ij} \) for each edge \( (i, j) \) as:

\[
T^*_i(x_i) = \prod_{j \in Ne(i)} m^*_{j \rightarrow i}(x_i)
\]

\[
T^*_{ij}(x_i, x_j) = \exp(\theta_{ij}(x_i, x_j)) \prod_{k \in Ne(i) \setminus j} m^*_{k \rightarrow i}(x_i) \prod_{l \in Ne(j) \setminus i} m^*_{l \rightarrow j}(x_j)
\]

These functions \( T^* \) define an alternative parameterization of the distribution \( p(\mathbf{x}; \theta) \) of Equation (1) as follows for both cyclic and acyclic graphs:

\[
p(\mathbf{x}; T^*) \propto \prod_{i \in V} T^*_i(x_i) \prod_{(i, j) \in E} T^*_{ij}(x_i, x_j)
\]

The following two properties hold for functions \( T^* \):

- If the graph is acyclic, then the functions \( T_i \) and \( T_{ij} \) are equivalent to the max-marginals \( P_i \) and \( P_{ij} \), which are defined as follows:

\[
P_i(x_i) = \kappa \max_{(\mathbf{x'}|x'_i=x_i)} p(\mathbf{x'}; \theta)
\]

\[
P_{ij}(x_i, x_j) = \kappa \max_{(\mathbf{x'}|x'_i=x_i, x'_j=x_j)} p(\mathbf{x'}; \theta)
\]

where \( \kappa \) is a constant that can be different for each node \( i \) and edge \( (i, j) \). Intuitively, \( P_i(x_i) \) is the probability of a most likely assignment \( \mathbf{x'} \) with \( x'_i \) fixed to \( x_i \) in \( \mathbf{x} \) and \( P_{ij}(x_i, x_j) \) is the probability of a most likely assignment with \( x'_i \) and \( x'_j \) fixed to \( x_i \) and \( x_j \) in \( \mathbf{x} \) respectively. Such reparameterization is helpful as the complete MAP assignment \( \mathbf{x}^* \) can then be extracted easily using one iteration of message-passing based on these max-marginals [31].

- If the graph is cyclic, then the functions \( T^* \) do not represent the true max-marginals. However, they satisfy certain local consistency requirements, and
they are commonly referred to as the pseudo-max-marginals. The approximate MAP assignment for such graphs can then be extracted from such pseudo-max-marginals and, as empirically demonstrated [1; 31], is quite close to the optimal MAP assignment.

Wainwright and Jordan also show that any positive distribution \( p(x) \) defined on a pairwise graph can be reparameterized in terms of pseudo-max-marginals [31]. Therefore, the fixed point of max-product updates exists for any arbitrary graph. Previously, this result was known only for acyclic graphs or graphs with only a certain number of cycles. This fundamental insight about the fixed point of max-product has resulted in the development of several successful and convergent message passing algorithms such as tree-reweighted max-product (TRMP) [30] and its variants [9].

5 Properties and Complexities of the Message Passing Algorithms

In this section, we discuss some of the desirable properties of approximate DCOP algorithms and specify whether the abovementioned algorithms have these properties.

5.1 Properties

- **Convergence**: An algorithm is said to have this property if it is guaranteed to converge to a fixed point after a finite number of iterations. For message passing algorithms, the fixed point is the point where the content of every message no longer changes. This property can be used as a basis to guarantee termination. The MPLP and EM algorithms have this property. The MS algorithm does not have this property unless it is operating on an acyclic graph. Most existing approximate DCOP algorithms like the \( k \)-, \( t \)- and \( C \)-optimal algorithms and those based on the Anytime Local Search framework have this property [16; 8; 29; 39].

- **Anytime**: An algorithm is said to have this property if it finds solutions whose qualities are monotonically non-decreasing. The EM algorithm has this property. Both the MPLP and MS algorithms do not have this property. Most existing approximate DCOP algorithms like the \( k \)-, \( t \)- and \( C \)-optimal algorithms and those based on the Anytime Local Search framework have this property [16; 8; 29; 39].

- **Error Bounded**: An algorithm is said to have this property if it has an error bound on the solution quality. The MPLP algorithm has this property. The upper bound \( UB \) can be calculated by using the \( \gamma \) messages as follows:

\[
UB = \sum_{i \in V} \max_{x_i} \sum_{k \in N e(i)} \gamma_{k \rightarrow i}(x_i)
\]  

(16)
Furthermore, this upper bound is also monotonically non-increasing. The EM and MS algorithms do not have this property. Most existing approximate DCOP algorithms like the $k$-, $t$- and $C$-optimal algorithms have this property but, unlike the MPLP upper bounds, their upper bounds are determined a priori before the start of the algorithm [16; 8; 29].

- **Dynamic**: An algorithm is said to have this property if it is able to solve dynamic DCOPs, that is, DCOPs that change over time. The MPLP, EM and MS algorithms have this property. These message passing algorithms can handle dynamic DCOPs easily since they can compute the content of the messages and update their marginal probabilities/beliefs in the exact same way whether a DCOP changes or not. On the contrary, other pseudotree-based DCOP algorithms like ADOPT and DPOP need to reconstruct their pseudotrees for the new problem before solving it. However, it is important to note that there are several extensions of these pseudotree-based algorithms to handle dynamic DCOPs as well [19; 20; 26; 37].

### 5.2 Complexity Analysis

In this section, we describe the space complexity of the algorithms and their messages. In each iteration, the MPLP, EM and MS algorithms send $|N_e(i)|$ number of messages for each node $i \in V$, resulting in a total of $2|E|$ number of messages. Each of these messages contain $k_i$ or, more generally, $O(k)$ floating point numbers, where $k_i$ is the domain size of the sending agent $x_i$ and $k$ is the maximum domain size. Therefore, the network load in each iteration is $O(|E|k)$.

Each agent in all three algorithms needs to store the contents of the messages received in the previous iteration, the contents of the messages to be sent out in the current iteration and the marginal probabilities/beliefs. Therefore, the memory requirement of each agent is $O(2|N_e(i)|k + k) = O(|V|k)$.

### 6 Experimental Results

We now report some preliminary experimental results that we have obtained. We evaluated MPLP in sensor network problems using the PEAV formulation [13; 36]. The targets are arranged in a grid and each target is surrounded by four sensors, all of which are needed to track the target. We used two problem sizes: problems with 16 targets arranged in a $4 \times 4$ grid and problems with 25 targets

<table>
<thead>
<tr>
<th>Instance</th>
<th>Time (sec.)</th>
<th>Cycles</th>
<th>Solution Quality</th>
<th>Error Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPLP</td>
<td>BnB-Adopt</td>
<td>MPLP</td>
<td>BnB-Adopt</td>
<td>MPLP</td>
</tr>
<tr>
<td>1</td>
<td>0.181</td>
<td>496.75</td>
<td>403</td>
<td>55940</td>
</tr>
<tr>
<td>2</td>
<td>0.183</td>
<td>584.59</td>
<td>308</td>
<td>65646</td>
</tr>
<tr>
<td>3</td>
<td>0.807</td>
<td>470.75</td>
<td>466</td>
<td>52947</td>
</tr>
<tr>
<td>4</td>
<td>0.236</td>
<td>502.00</td>
<td>467</td>
<td>56613</td>
</tr>
<tr>
<td>5</td>
<td>0.160</td>
<td>484.32</td>
<td>382</td>
<td>54523</td>
</tr>
</tbody>
</table>

Table 1. Results for $4 \times 4$ grids
<table>
<thead>
<tr>
<th>Instance</th>
<th>Time (sec.)</th>
<th>Cycles</th>
<th>Solution Quality</th>
<th>Error Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.502</td>
<td>469</td>
<td>4773</td>
<td>0.06</td>
</tr>
<tr>
<td>2</td>
<td>0.685</td>
<td>697</td>
<td>5034</td>
<td>0.16</td>
</tr>
<tr>
<td>3</td>
<td>0.807</td>
<td>1087</td>
<td>5025</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.477</td>
<td>491</td>
<td>5279</td>
<td>1.13</td>
</tr>
<tr>
<td>5</td>
<td>0.433</td>
<td>447</td>
<td>5078</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 2. Results for $5 \times 5$ grids

arranged in a $5 \times 5$ grid. The constraint costs are generated randomly. We also ran BnB-Adopt [36], an optimal DCOP search algorithm, as a baseline comparison.

We conducted our experiments on a machine with 8GB of RAM and 2.66GHz CPU. We measured the runtimes of the algorithms in the number of seconds and the number of (synchronous) cycles [15]. Table 1 shows the results for the $4 \times 4$ grids with 16 targets. The table shows that MPLP terminates significantly faster, in both seconds and cycles, compared to BnB-Adopt and still yields an optimal solution. The last column indicates the difference between the lower bound on the cost and the actual cost. Table 2 shows the results of MPLP for the $5 \times 5$ grids with 25 targets. Again it shows that MPLP finds near optimal solutions and the error bound is very small. We did not report the results for BnB-Adopt because the algorithm failed to terminate within a time limit of 2 hours.

The tables show that MPLP can find solutions with very small error bounds for both problem sizes and finds them by several orders of magnitude faster than BnB-Adopt. Although these results are preliminary, we believe that they demonstrate the potential of message passing algorithms to solve DCOPs.

7 Conclusions

Researchers in the machine learning community have long studied the maximum a posteriori (MAP) estimation problem because of its application in problems like computer vision and bioinformatics. On the other hand, researchers in the multi-agents community have studied the distributed constraint optimization problem (DCOP) because of its application in multi-agent coordination problems like sensor networks. However, both of these independently formulated problems bear strong similarities. For example, both problems are graphical models.

In this paper, we formally showed the similarities between these two problems and described three message passing algorithms, namely max-product linear programming (MPLP), expectation maximiation (EM) and max-sum (MS), that were developed to solve the MAP estimation problem. We also showed that these algorithms can be easily tailored to solve DCOPs as well. We demonstrated the feasibility of this approach with a preliminary set of experiments, where we evaluated MPLP on sensor network problems. The results showed that MPLP can find solutions with very small error bounds and finds them by several orders of magnitude faster than BnB-Adopt, an optimal DCOP search algorithm. Therefore, we believe that they demonstrate the potential of message passing algorithms to solve DCOPs.
Bibliography


SOCIAL DCOP - Social Choice in Distributed Constraints Optimization

Arnon Netzer\textsuperscript{1} and Amnon Meisels\textsuperscript{1}

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
\{netzerar,am,alongrub\}@cs.bgu.ac.il

Abstract. Distributed Social Constraints Optimization Problems (DSCOPs) are DCOPs in which the global objective function for optimization incorporates a social welfare function (SWF). DSCOPs have individual, non-binary and asymmetric constraints and thus form a unique DCOP class. DSCOPs provide a natural framework for agents that compute their costs individually and are therefore self-interested. The concept of social welfare is discussed and SWFs are presented. An important aspect of DSCOPs and of social objective functions is their ability to support distributed hill climbing algorithms. The DSCOP hill climbing algorithm is presented and tested experimentally on multi agent pickup and delivery problems. It turns out to improve the distribution of utility gains among agents, while losing very little in global gain.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) form a powerful framework for distributed problem solving that has a wide range of application in Multi Agent Systems. Typical examples include mobile sensing networks\cite{10}, graphical games \cite{12} and synchronization of traffic lights\cite{8}. In a standard DCOP agents need to assign values in a way that would minimize (or maximize) the sum of the resulting constraints costs. A DCOP algorithm searches the combinatorial assignments space, in order to find the combination of assignments that will produce this minimum \cite{14, 15, 6}.

Consider a multi agent system in which each agent needs to solve its own Pickup and Deliver Problem (PDP). Packages need to be picked up and delivered under a set of conditions, each package adds a fixed utility, and every distance traveled adds some given cost (for extensive surveys on PDPs see \cite{1}).

In some cases an agent will be better off if another agent will pick one of the packages assigned to it. Figure 1 shows a two agent system. The packages of one agent are marked by small triangles and of the other agent by small circles. It is easy to see that all of the triangle packages are located at the left side of the area and all of the circled packages except one are on the right hand side. This may lead to a situation where the added trip to reach the leftmost package, for the circled agent, may cost more than its gain.

Since each package must be picked up, an agent can only discard a package if another agent picks it up. The problem is that no cost (or utility) can be assigned directly
to the package. The amount of utility change that arises from the transfer of a package from one agent to another depends on the other packages that need to be picked up by that agent. In fact, calculating this utility may be a non trivial Planning problem, that needs to be solved by the receiving agent.

![Fig. 1. A two agents Pickup and Deliver problem](image)

Further more, agents that transfer a package from one to the other cannot be only motivated by the global objective of minimizing or maximizing the sum of all utilities. This aspect of the standard DCOP model seems to be unrealistic. A more realistic approach would be to use an objective function that incorporates some individual gains of the participating agents.

The present paper proposes a DCOP model that attempts to answer the above two problems. It generalizes the standard DCOP model and addresses self-interested agents. The proposed model discards two common assumptions of the standard DCOP model. First, it assigns costs or utilities to states and not to constraints. Second, it replaces the standard objective function that minimizes the sum of all costs by a Social Welfare function that incorporate fairness and equality measures.

In Distributed Social Constraints Optimization Problems (DSCOPs) every agent autonomously computes its utility, based on its assignment and on the assignments of agents it is constrained with. The individual costs and gains of agents represent self-interested agents that may not find it natural to search cooperatively for a solution that maximizes the sum of all utilities as a target function.

Recently [3] proposed the use of Leximin as a target function for DCOPs, attempting to increase fairness. Their discussion was limited to DCOPs with hard constraints and unary costs only. Further more, the Leximin SWF may not not be the best SWF for such purpose [16].

The proposed model of DSCOPs uses target functions that incorporate both efficiency and equality. The present study proposes objective functions that arise from known social welfare functions (SWFs) that have been studied extensively in the field.
of Economics (cf. [2, 20]). These ideas are discussed in Section 3 and some of the key Social Welfare Functions (SWF) are reformulated for multi agent systems.

A social local search algorithm for DSCOPs is presented in section 4. The behavior of the proposed framework is tested on an experimental framework of truck delivery problems (Section 5).

2 Distributed Social Constraint Optimization Problems

DSCOPs are based on the following assumptions:

– Each agent can compute its utility based on its assignment and the assignments of its constraining agents.
– The method of computation of utility is private to each agent.
– Cooperation is expressed by a faithful search of agents for the optimum of the objective function.

More formally, a DSCOP is a tuple $\langle A, \mathcal{X}, D, \mathcal{N} \rangle$. $A$ is a finite set of agents $A_1, A_2, ..., A_n$. $\mathcal{X}$ is a finite set of variables $X_1, X_2, ..., X_m$. Each variable is held by a single agent (an agent may hold more than one variable). $D$ is a set of domains $D_1, D_2, ..., D_m$. Each domain $D_i$ contains a finite set of values which can be assigned to variable $X_i$. $\mathcal{N}$ is a list of neighbors $N_1, N_2, ..., N_m$. Each list $N_i$ contains a list of neighbors $n_1, n_2, ..., n_k$, which includes all the agents that influence the utility evaluation of agent $A_i$.

An assignment is a pair, a variable and a value from that variable’s domain. A complete assignment is an assignment for every variable in $\mathcal{X}$. A utility profile is the collection of all the utilities calculated by the agents for a given complete assignment. An optimal solution is a complete assignment whose utility profile optimizes the objective function.

The DSCOP model can be viewed as a generalization of Asymmetric DCOPs (AD-COPs) [7]. ADCOPs incorporate a different cost for each agent that is involved in a constraint. Unlike ADCOPs, the proposed model of DSCOPs does not assume the costs of constraints of any given agent to be additive. The underlying framework of distributed social COPs (DSCOPs) does not impose any restriction on the form of computation of the utility of each agent. All costs of an assignment of a DSCOP are represented as individual costs (utilities) of individual agents in a given compound assignment.

Consider the distributed delivery example of section 1 (Figure 1). One can represent it by a K-ary constraint associated with every given package that enforces the pickup of the package by some agent. The utility change for a given agent, due to the addition or the removal of a package depends on the other packages that need to be picked up by that agent. The utility also depends on parameters of the PDP of the specific agent (number of trucks, sizes of trucks, etc.).

Different assignments of packages result in different PDPs for each agent. A solution of the problem is a redistribution of the packages between the agents, that will optimize a given target function on the utilities calculated by solving the PDPs of individual agents.
Interestingly, this type of problem was addressed in [17] in the form of a DCOP. The DCOP of [17] defines the domain of each agent to be all possible combinations of packages it can pick up and includes constraints to ensure the delivery of all packages. Such a representation has domains of size that is exponential in the number of packages that can be picked up by an agent. It also does not take into account the self interest of the agents (e.g., it maximizes the sum of all personal utilities).

An agent who travels a lot and picks up many packages is more likely to gain from picking up additional packages. In contrast, an agent that picks up a small number of packages may have a smaller probability of traveling near the new package. Agents that are likely to lose have little or no incentive to cooperate in search for the optimal assignment that maximizes the sum of all gains.

The DSCOP model enables the use of alternative goal (or objective) functions, that increase fairness and equality among the utilities of different agents. The view of the present paper is that fairness, or the equality of gains of individual agents, is the key for securing the cooperation of agents in search for an optimal solution.

3 Social Welfare Functions

The Social Welfare Function (SWF) was introduced by Abram Bergson [2] and is used as a way to rank the possible states of a society. For a DCOP, a SWF can serve as the objective function that the agents are optimizing.

**Definition 1.** Given a finite set of agents \( \mathcal{A} = (a_1, ..., a_n) \), \( u_i = \text{utility}(a_i) \), and a utility profile \( \mathcal{U} = (u_1, ..., u_n) \), a Social Welfare Function maps \( \mathcal{U} \) to the real numbers: \( \text{SFW}(\mathcal{U}) \to \mathbb{R} \).

The objective function used in standard DCOPs [15] is the max-sum (or min-sum) which is the Utilitarian SWF. The Utilitarian SWF is an extreme example in which only the state of the whole group of agents is considered, with no consideration for the welfare of any specific agent. This may lead to situations in which the sum of all the costs of all the agents is minimal but specific agents carry an un-proportionate part of the cost. For example, according to the Utilitarian SWF a cost distribution between two agents of \( (u_1 = 100, u_2 = 0) \) has the same score as a cost distribution of \( (u_1 = 50, u_2 = 50) \).

The Egalitarian SWF (max min) [9] forms another extreme, in which only the welfare of the worst well-off (the poorest) agent is considered. The Egalitarian SWF makes sure that the worst-off agent will be in as good a state as it can be, but, it does not reflect the situation of all the other agents. For example, according to the Egalitarian SWF a cost distribution between two agents of \( (u_1 = 100, u_2 = 0) \) has the same score as a cost distribution of \( (u_1 = 50, u_2 = 0) \).

3.1 Pareto Optimality

A fundamental concept in social welfare is Pareto optimality. Given two utility profiles, one is said to be Pareto better than the other if at least one agent prefers the former, and no agent prefer the latter [18].
Definition 2. Given a finite set of agents \( A = (a_1, ..., a_n) \), a utility profile \( U = (u_1, ..., u_n) \) is Pareto better than \( U' = (u'_1, ..., u'_n) \) iff \( \forall i : u_i \geq u'_i \) and \( \exists j : u_j > u'_j \).

A utility profile is Pareto optimal if no other utility profile is Pareto better than it. The collection of all the Pareto optimal utility profiles is the Pareto frontier. Note that Pareto optimality of a utility profile is a characteristic of the profile itself, and is independent of any specific SWF. It is desirable that a SWF would prefer a Pareto better utility profile over a Pareto inferior one.

It is easy to see that the Utilitarian SWF is Pareto optimal, since an increase in an agent utility, ceteris paribus, will increase the sum of all agents utilities. The Egalitarian SWF acts differently than the utilitarian SWF. An increase in the utility of an agent that is not the worst off, will not change the SWF evaluation of the utility profile. An adaptation of the Egalitarian SWF called Leximin [5] solves this problem.

3.2 Fairness and equality in SWF

The basic property for introducing fairness into a SWF is the Pigou-Dalton transfer principle [19, 4]. According to this principle, a transfer of utility between one agent and another is desirable as long as it decreases the difference between the utilities of the two agents, making them more equal.

Definition 3. A SWF satisfies the Pigou-Dalton transfer principle if given a finite set of agents \( A = (a_1, ..., a_n) \), \( u_i = utility(a_i) \), a utility profile \( U = (u_1, ..., u_n) \), and \( U' = (u'_1, ..., u'_n) \), \( \forall k \neq i, j : u_k = u'_k \cdot u_i + u_j = u'_i + u'_j \) and \( |u_i - u_j| > |u'_i - u'_j| \) then \( U' \) is preferred over \( U \).

It is easy to see that neither the Utilitarian SWF nor the Egalitarian SWF (or Leximin) satisfy the Pigou-Dalton transfer principle. Perhaps the most straightforward SWF that does satisfy the Pigou-Dalton transfer principle is the Nash SWF [13]. Which is the multiplication of all agents utilities \( SWF_{Nash} = \prod_{i=1}^{n} u_i \).

Though the Nash SWF is Pareto optimal and satisfies the Pigou-Dalton transfer principle, it is considered to be very biased toward making the poorest better off, at the expense of the total well being of the society.

Sen [20] defined a family of Social Welfare Functions that incorporate both the inequality and the efficiency measures. In particular the Social Welfare Function based on the Gini index [21] \( W_{Gini} = \mu(1 - G) \) where \( \mu \) is the average utility and \( G \) is the Gini index. Another example is the Theil based SWF: \( W_{Theil} = \mu * e^{-T} \) where \( T \) is the Theil Index \( T = \frac{1}{N} \sum_{i=1}^{N} \frac{x_i}{\bar{x}} \cdot ln \frac{x_i}{\bar{x}} \). The present paper proposes the use of such a SWF for DSCOPs, in order to attain the right balance between the self interest of agents and the global interest (e.g. efficiency).

4 Hill climbing for DSCOP

Any hill climbing local search algorithm checks whether an assignment change will increase the overall evaluation of the objective function. In standard DCOP it is enough
to ensure that when an agent replaces its assignment, none of its neighbors will replace their assignments at the same time [22, 7].

Consider the example in Figure 2. In this example every circle is an agent and every line denotes a constraint between two agents. Assume that after each agent computed its best possible assignment change (possibly using additional information from its neighbors) agents A and C discover that they can make the best assignment change out of all their neighbors. Due to the additivity of constraint costs in standard DCOPs, there is no problem for agents A and C to replace their assignment at the same time.

This is no longer true for DSCOPs, due to the fact that utilities in DSCOP are calculated by each agent and the utility is not necessarily additive. Consider agent B in Figure 2. It is possible that both an assignment change by A and an assignment change by C will decrease B’s utility.

For DSCOPs it is not enough to ensure that none of an agents neighbors are changing assignment if the agent does. One must make sure that none of the neighbor-neighbors change their assignment as well. In the example of Figure 2 that means that agents A and D can move simultaneously, but agents A and C cannot.

Figure 3 presents the pseudo code for a cycle-based hill climbing local search algorithm for solving DSCOPs. The proposed algorithm is independent of the specific objective function. This algorithm will be slightly adapted for the specific DSCOP of the experimental evaluation in section 5.

The proposed algorithm in Figure 3 is a greedy hill climber. Each step taken by an agent not only improves the global target function evaluation, it is also guaranteed to be the “best” possible step in its neighborhood, as in the Max Gain Message (MGM) algorithm [11].

The algorithm includes a preprocess stage in which all agents construct a list of all their second order neighbors. This list includes all of their neighbors and all of their neighbors’ neighbors (lines 1–4 in pre-process). The list is termed $b_{neighbors}$.

At the beginning of the main procedure (line 1), each agent selects an assignment. Since the assignments of neighbors are unknown at this stage, the assignment is selected
by considering only the unary part of the utility. This assignment is then sent to all neighbors, and all neighbors assignments are received (lines 2,3).

The termination condition (line 4) may be a given number of cycles, or some threshold on progress. Until the termination condition is met all the following steps are repeated.

Each agent calculates its utility according to its assignment and its neighbors assignments (line 5). Each agent selects its best assignment (line 6) taking into account the assignments of all its neighbors. A new utility is calculated assuming no assignment changes in the neighbors (line 7). This assignment is sent to all neighbors (line 8) and the proposed assignments are collected (line 9). For every neighbor that wants to replace its assignment, the agent computes its new utility assuming only this neighbor will change its assignment (line 11). This utility is sent back to the specific neighbor (line 12).

The utility reports of all neighbors are calculated based on the agent’s proposed assignment change (assuming it is the only one changing its assignment in the neighborhood - line 13). If the new assignment improves the local evaluation of the target function (line 14), the agent sends the amount of improvement (line 15) to all its b_neighbors list.

The agent with the largest improvement makes the move by updating its assignment and utility (lines 19,20).

5 Experimental Evaluation

Experiments where performed on a multi agent system in which every agent has a simple PDP. Each agent has one truck with limitless capacity that needs to get out of the agent offices, travel through all packages and return to the agent offices (see section 2).

This problem needs a slightly modified algorithm from the generic one described in 4. This is because an agent cannot decide unilaterally on an assignment change. In the present case an assignment change means that an agent is interested in transferring a package to another agent and it must find the other agent that will take this package.

5.1 DSCOP for PDPs

Figure 4 presents the hill climbing local search algorithm for the multi agent PDP system. The preprocess stage of the algorithm is identical to the general one (in Figure 3), and results in the b_neighbor list of each agent.

The agents then check a termination condition (line 4) which may be a given number of cycles, or the detection whether progress is below a given threshold. Until the termination condition is met all following steps are repeated.

Utility is calculated by solving the PDP with the packages owned by the agent (line 3). The utility is sent to all neighbors (line 4), and all neighbors utilities are collected (line 5). The agent finds if and what package it wants to hand over to someone else by solving the PDPs for all options of removing a single package from the packages it owns (line 6). This package is offered to all relevant neighbors (line 7).
pre-process:
1. send Neighbors list to all neighbors
2. collect Neighbors list from all neighbors
3. \( b_{neighbors} \leftarrow \) all neighbors
4. \( b_{neighbors} \leftarrow \) all neighbors neighbors

Main Procedure:
1. \( current\_assignment \leftarrow \) ChooseBestAssignment()
2. send \( current\_assignment \) to neighbors
3. collect neighbors assignments
4. while (no termination condition is met)
5. utility \( \leftarrow \) CalcUtility(current\_assignment)
6. assignment \( \leftarrow \) ChooseBestAssignment()
7. utility\_new \( \leftarrow \) CalcUtility(assignment)
8. send assignment to neighbors
9. collect neighbors assignments
10. foreach (neighbor \( n \))
11. \( u_n \leftarrow \) utility assuming new \( n \) assignment
12. send \( u_n \) to neighbor \( n \)
13. collect neighbors utilities based on assignment
14. if assignment improves Target Function
15. utility\_diff \( \leftarrow \) utility \( \) utility\_new
16. send utility\_diff to all \( b_{neighbors} \)
17. collect utility\_diff from all \( b_{neighbors} \)
18. if utility\_diff > utility\_diffs off all \( b_{neighbors} \)
19. \( current\_assignment \leftarrow \) assignment
20. utility \( \leftarrow \) utility\_new

Fig. 3. Greedy Hill Climbing for DSCOPs

For each package that was offered to a given agent, that agent calculates the resulting potential gain (line 10). The computed gain is sent back to the offering agent (line 11).

Each agent collects all the responses (line 12) and decides which (if any) of the neighbors will get the package (line 13). The agent then calculates the gain (according to the target function) from transferring the package to the selected neighbor (line 14), and sends this gain to all \( b_{neighbors} \) (line 15).

Agents collect the potential gains from all \( b_{neighbors} \) (line 16). The offered package is removed from the agent’s package list (line 18) and sent to the selected neighbor whose added gain is largest (line 19). The selected agent gets a message and adds the package to its package list (line 22).

5.2 Experimental Setup and Results

The setup for the experiment includes an area represented by a grid of 100X100 in which 150 agents are randomly placed. Each agent has a single truck and is given an area defined by a circle centered at the agent location with a Radius of 10. Each agent picks up packages within its circle, and assigned five packages that are randomly placed
Main Procedure:
1. current\_assignment ← all packages belonging to this agent
2. while (no termination condition is met)
   3. utility ← SolvePDP(current\_assignment)
   4. send utility to neighbors
   5. collect neighbors utilities
   6. suggested\_package ← FindWhatPackageToGive()
   7. send suggested\_package to all neighbors that can pick it
   8. collect neighbors suggested packages
9. foreach (neighbor n)
   10. u\_n ← SolvePDP(current\_assignment + Package\_n)
   11. send u\_n to neighbor n
   12. collect neighbors responds to suggested\_package
   13. choose best neighbor to give the package to.
   14. utility\_diff ← improvement due to the package transfer.
   15. send utility\_diff to all b\_neighbors.
   16. collect utility\_diff from all b\_neighbors
   17. if utility\_diff > utility\_diffs off all b\_neighbors
      18. remove suggested\_package from current\_assignment
      19. send suggested\_package to selected neighbor
   20. else
   21. if received Package from neighbor
   22. add Package to current\_assignment

Fig. 4. DSCOP for the PDP problem

inside its area. The Pickup and Delivery Problems were solved using a straight forward $A^*$ algorithm. The results were accumulated and averaged over 10 randomly generated problems.

Each problem was solved with two different objective functions. MaxSum (utilitarian) and the Theil based SWF. Table 1 presents the comparison of the MaxSum and Theil SWF with respect to three different attributes: The averaged utility, the Theil Index and the Egalitarian utility.

<table>
<thead>
<tr>
<th></th>
<th>Theil SWF</th>
<th>MaxSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Utility</td>
<td>32.4467</td>
<td>33.4996</td>
</tr>
<tr>
<td>Theil Index</td>
<td>0.03455</td>
<td>0.1199</td>
</tr>
<tr>
<td>Egalitarian utility</td>
<td>13.406</td>
<td>1.8407</td>
</tr>
</tbody>
</table>

Table 1. Theil SWF vs. MaxSum (utilitarian)

It is easy to see that for a small sacrifice in the overall average utility (less then 3.5\%), one can get an improvement by a factor of 3.5 in the Theil index (a Theil index closer to 0 represents a more equal solution). This means that the utility is spread more
evenly among the agents. The Egalitarian utility is also improved by a factor of 7.2. That means that the “worst off” agent is in a much better state.

The above results can be further illustrated by inspecting the histogram of the utilities of the individual agents (Figure 5). It is clear that when using MaxSum the histogram has a much wider spread. Setting the Theil SWF as the target function results in a much narrower histogram, illustrating the higher level of equality in the distribution of utility among agents.

![Figure 5. Histogram of the utility of Agents](image)

Figure 6 is a histogram of the number of packages that are assigned to each agent at the end of the algorithm execution. Again we see the same pattern of having a much more centered histogram when using Theil SWF as the target function.

![Figure 6. Histogram of the number of Packages per Agent](image)

The use of MaxSum as the objective function results in some agents picking up a large number of packages. Since the solution complexity of the PDP in the general case is exponential in the number of packages, solving the problem with MaxSum as the target function results in a much higher run time than using the Theil SWF as a target function. In fact, this type of problem took over a hundred times more CPU time for MaxSum than for the Theil SWF.
Figure 7 shows a histogram of the utility gain of agents. The use of MaxSum produces many low gains. In contrast, Theil SWF produces a relatively small number of agents that lose utility.

![Histogram of utility gain per agent](image)

**Fig. 7.** Histogram of the utility gain per Agent

In order to see the effect of the different target functions when the initial state of the agents is unbalanced, a test was performed with a nonuniform number of packages per agent. 120 agent were assigned 4 packages each (small agents), and 30 were assigned 9 (large agents). The results are in Table 2

<table>
<thead>
<tr>
<th></th>
<th>Theil SWF</th>
<th>MaxSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Gain of large agents</td>
<td>-2.054</td>
<td>12.932</td>
</tr>
<tr>
<td>Average Gain of small agents</td>
<td>6.021</td>
<td>3.312</td>
</tr>
</tbody>
</table>

**Table 2.** Agents gains for non uniform initial state

Using MaxSum as target function results in each of the large agents gaining, on the average, almost four times the amount of utility gained by each the small agents. A large agent that travels more, has a higher probability of gaining from adding a package, due to higher probability of traveling near its location. When using the Theil based SWF, the effect is reversed. The large agents lose utility, while the small ones are gaining almost twice the amount they gained in the MaxSum scenario. This demonstrates the equality enhancing characteristic of the Theil based SWF.

### 6 Conclusions

A new framework - Distributed Social Constraints Optimization Problems (DSCOP) - is introduced for multi agent cooperation in a constrained problem. In DSCOPs each agent calculates its utility based on its assignment and the assignments of all of its constraining agents. Individual utilities, based on states and computed by each agent
naturally represent self-interested agents. The cooperation of such agents in finding an optimal solution for the DSCOP needs to be based on an attractive social choice.

A reasonable social choice can be based on the optimization of objective functions that incorporate equality index based Social Welfare Functions. This idea is investigated and a simple objective function is proposed - the Theil index.

The present study designs a hill climbing algorithm for DSCOPs that is based on the Theil index as objective function. In order to investigate the behavior of DSCOPs and the proposed local search method, experiments were performed on a multi agent Pickup and Delivery Problem (PDP). In these problems each agent has its own PDP to solve. It turns out that cooperation between agents (hanging over unwanted packages to other agents) can improve the utility of agents.

The experiments demonstrated that for a small trade-off in the overall utility, the use of the Theil based SWF objective function achieves a much more attractive solution. The distribution of utility among agents becomes more uniform and the agent with the lowest utility (Egalitarian) has a much higher utility than in the solution of the same problem with the MaxSum objective function.

Experiments with a non uniform initial distribution of agents’ wealth demonstrated the tendency to become less equal when optimizing for MaxSum, making the “rich” richer. Using the Theil SWF on the other hand, had the opposite effect. It lowered the utility of the “rich” agents and increased the utility of the smaller (“poorer”) ones, arriving at efficiency of 95 percent.

References


Anytime Exploration for Distributed Constraint Optimization

Hilla Peled and Roie Zivan
Industrial Engineering and Management department, Ben Gurion University, Beer-Sheva, Israel
{hillapel,zivanr}@bgu.ac.il

Abstract. Distributed Constraint Optimization Problems (DCOPs) are an elegant model for representing and solving many realistic combinatorial problems which are distributed by nature. DCOPs are NP-hard and therefore most recent studies consider incomplete (local) search algorithms for solving them. Distributed local search algorithms can be used for solving DCOPs. However, because of the differences between the global evaluation of a system’s state and the private evaluation of states by agents, agents are unaware of the global best state which is explored by the algorithm. Previous attempts to use local search algorithms for solving DCOPs reported the state held by the system at the termination of the algorithm, which was not necessarily the (global) best state explored.

A general framework for implementing distributed local search algorithms for DCOPs was proposed in [24]. The framework makes use of a BFS-tree in order to accumulate the costs of the system’s state in its different steps and to propagate the detection of a new best step when it is found. The resulting framework enhances local search algorithms for DCOPs with the anytime property. However, since most local search algorithms are mostly exploitive the potential of the anytime framework has not been explored.

We propose a set of increased exploration heuristics that exploit the proposed anytime framework. Our empirical study reveals the advantage of the use of the proposed heuristics in the anytime framework over state of the art local search algorithms.

1 Introduction

The Distributed Constraint Optimization Problem (DCOP) is a general model for distributed problem solving that has a wide range of applications in Multi-Agent Systems and has generated significant interest from researchers [1, 8, 11, 13, 15, 22]. DCOPs are composed of agents, each holding one or more variables. Each variable has a domain of possible value assignments. Constraints among variables (possibly held by different agents) assign costs to combinations of value assignments. Agents assign values to their variables and communicate with each other, attempting to generate a solution that is globally optimal with respect to the costs of the constraints [11, 12].

There is a wide scope in the motivation for research on DCOPs, since they can be used to model many every day combinatorial problems that are distributed by nature.
Some examples are the Nurse Shifts assignment problem [16, 6], the Sensor Network Tracking problem [22], and Log Based Reconciliation problem [2].

DCOPs represent real life problems that cannot or should not be solved centrally for several reasons, among them lack of autonomy, single point of failure, and privacy of agents.

A number of studies on DCOPs presented complete algorithms [11, 13, 4]. However, since DCOPs are NP-hard, there is a growing interest in the last few years in local (incomplete) DCOP algorithms [7, 22, 24, 17, 19]. Although local search does not guarantee that the obtained solution is optimal, it is applicable for large problems and compatible with real time applications.

The general design of most state of the art local search algorithms for DCOPs is synchronous (DALO is the only published exception). In each step of the algorithm an agent sends its assignment to all its neighbors in the constraint network and receives the assignment of all its neighbors. They differ in the method agents use to decide whether to replace their current value assignments to their variables, e.g., in the max gain messages algorithm (MGM) [7], the agent that can improve its state the most in its neighborhood replaces its assignment. A stochastic decision whether to replace an assignment is made by agents in the distributed stochastic algorithm (DSA) [22].

In the case of centralized optimization problems, local search techniques are used when the problems are too large to perform a complete search. Traditionally, local search algorithms maintain a complete assignment for the problem and use a goal function in order to evaluate this assignment. Different methods which balance between exploration and exploitation are used to improve the current assignment of the algorithm [5, 14, 18]. An important feature of most local search algorithms is that they hold the best assignment that was found throughout the search. This makes them anytime algorithms, i.e., the quality of the solution can only increase or remain the same if more iterations of the algorithm are performed [23]. This feature cannot be applied in a distributed environment where agents are only aware of the cost of their own assignment (and maybe their neighbors too) but no one actually knows when a good global solution is obtained.

In [24] a general framework for enhancing local search algorithms for DCOPs which follows the general synchronous structure with the anytime property, was proposed. In the anytime local search framework for DCOPs (ALS\_DCOP), the quality of each state is accumulated via a Breadth First Search tree (BFS-tree) structure. Agents receive the information about the quality of the recent states of the algorithm from their children in the BFS-tree, calculate the resulting quality including their own contribution according to the goal function, and pass it to their parents. The root agent makes the final calculation of the cost for each state and propagates down the tree the index number of the most successful state. When the search is terminated, all agents hold the assignment of the best state according to the global goal function.

In order to produce the best state out of $m$ steps, the algorithm must run $m + (2 \cdot h)$ synchronous steps where $h$ is the height of the tree used. ALS\_DCOP does not require agents to send any messages beside the messages sent by the original algorithm. The space requirements for each agent are $O(h)$ and it preserves a high level of privacy (see [24] for details).
In this paper we study the potential of the proposed framework by proposing a set of exploration methods (heuristics) which exploit the anytime property by introducing extreme exploration to exploitive algorithms. We present an intensive empirical evaluation of the proposed methods on three different benchmarks for DCOPs. The proposed methods find solution of higher quality than state of the art algorithms when implemented within the anytime local search framework.

The rest of the paper is organized as follows: Section 2 describes the distributed constraint optimization problem (DCOP). State of the art local search algorithms for solving DCOPs, will be presented in Section 3. Section 4 presents ALS_DCOP. Section 5 presents a set of heuristics which increase the exploration of standard local search algorithms. We evaluate the performance of the proposed heuristics in Section 6 Section 7 presents our conclusions.

2 Distributed Constraint Optimization

A DCOP is a tuple \( \langle A, X, D, R \rangle \). \( A \) is a finite set of agents \( A_1, A_2, ..., A_n \). \( X \) is a finite set of variables \( X_1, X_2, ..., X_m \). Each variable is held by a single agent (an agent may hold more than one variable). \( D \) is a set of domains \( D_1, D_2, ..., D_m \). Each domain \( D_i \) contains the finite set of values which can be assigned to variable \( X_i \). \( R \) is a set of relations (constraints). Each constraint \( C \in R \) defines a non-negative cost for every possible value combination of a set of variables, and is of the form \( C : D_{i_1} \times D_{i_2} \times ... \times D_{i_k} \rightarrow \mathbb{R}^+ \cup \{0\} \). A binary constraint refers to exactly two variables and is of the form \( C_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\} \). A binary DCOP is a DCOP in which all constraints are binary. An assignment (or a label) is a pair including a variable, and a value from that variable’s domain. A partial assignment (PA) is a set of assignments, in which each variable appears at most once. \( \text{vars}(PA) \) is the set of all variables that appear in \( PA \), \( \text{vars}(PA) = \{X_i \mid \exists a \in D_i \land (X_i, a) \in PA\} \). A constraint \( C \in R \) of the form \( C : D_{i_1} \times D_{i_2} \times ... \times D_{i_k} \rightarrow \mathbb{R}^+ \cup \{0\} \) is applicable to \( PA \) if \( X_{i_1}, X_{i_2}, ..., X_{i_k} \in \text{vars}(PA) \). The cost of a partial assignment \( PA \) is the sum of all applicable constraints to \( PA \) over the assignments in \( PA \). A full assignment is a partial assignment that includes all the variables (\( \text{vars}(PA) = X \)). A solution is a full assignment of minimal cost.

In this paper, we will assume each agent owns a single variable, and use the term “agent” and “variable” interchangeably. We will assume that constraints are at most binary and the delay in delivering a message is finite [11, 21]. Agents are aware only of their own topology (i.e. only of their own neighbors in the constraints network and the constraints that they personally and privately hold).

3 Local Search for Distributed Constraints problems

The general design of most state of the art local search algorithms for Distributed Constraint Satisfaction and Optimization Problems (DisCSPs and DCOPs) is synchronous. In each step of the algorithm an agent sends its assignment to all its neighbors in the constraint network and receives the assignment of all its neighbors. We present as an example an algorithm that applies to this general framework, the Distributed Stochastic Algorithm (DSA) [22]. The algorithms is presented following the
recent version of [22]. Notice that these algorithms were first designed for distributed satisfaction problems in which a solution must not violate any constraint, but they can be applied as is to Distributed Max CSPs (where the optimal solution is the complete assignment with the smallest number of violated constraints) which is a special type of DCOPs. Thus in our description we consider an improvement a decrease in the number of violated constraints (as in Max-CSPs).

The basic idea of the DSA algorithm is simple. After an initial step in which agents pick some value for their variable (random according to [22]), agents perform a sequence of steps until some termination condition is met. In each step, an agent sends its value assignment to its neighbors in the constraints graph and receives the assignments of its neighbors. After collecting the assignments of all its neighbors, an agent decides whether to keep its value assignment or to change it. This decision which is made stochastically has a large effect on the performance of the algorithm. According to [22], if an agent in DSA cannot improve its current state by replacing its current value, it does not replace it. If it can improve, it decides whether to replace the value using a stochastic strategy (see [22] for details on the possible strategies and the difference in the resulting performance). A sketch of DSA is presented in Figure 1. After a random value is assigned to the agent’s variable (line 1) the agent performs a loop (each iteration of the loop is a step of the algorithm) until the termination condition is met. In each step the agent sends its value assignment to all its neighbors and collects the assignments of all its neighbors (lines 3, 4). According to the information it receives, it decides whether to replace its assignment; when the decision is positive it assigns a new value to its variable (lines 5, 6).

**DSA**

1. `value ← ChooseRandomValue()`
2. `while` (no termination condition is met)
3. `send value to neighbors`
4. `collect neighbors’ values`
5. `if` (ReplacementDecision())
6. `select and assign the next value`

**Fig. 1.** Standard DSA for DisCSPs.

An example of a DisCSP is presented in Figure 2. Each of the agents has a single variable with the values a and b in its domain. Dashed lines connect constrained agents and all constraints are equality constraints. Although DSA is a uniform algorithm, i.e., the algorithm does not assume the existence of agents’ identifiers, we added identifiers to the figure to make it easier to describe.

Before the first step of the algorithm each agent selects a random number. Assume agents 1, 3, and 5 selected a and agents 2 and 4 selected b. In the first step all agents can improve their states by changing their assignment. Following a stochastic decision

---

1 In this paper we follow the general definition of a DCOP and a DisCSP which does not include a synchronization mechanism. If such a mechanism exists, agents in DSA can send value messages only in steps in which they change their assignments.
only agents 2 and 5 replace their assignment. Now agents 1, 2, and 3 hold \( a \) and agents 4 and 5 hold \( b \). At this step only agent 4 can replace its assignment and in the next only agent 5 can replace. In the resulting state, all agents are holding \( a \) and the algorithm is terminated.

4 Anytime Local Search framework for DCOPs

The ALS\_DCOP framework enhances DCOP synchronous local search algorithms with the \textit{anytime} property [24]. In the proposed framework, ALS\_DCOP, a tree is used as in ADOPT [11] and DPOP [13]. In contrast to ADOPT and DPOP that require the use of a pseudo-tree, the only requirement in ALS\_DCOP is that every agent has a parent route to the root agent. Thus, a Breadth First Search (BFS) tree on the constraint graph can be used. The BFS-tree structure is used in order to accumulate the cost of agents assignments in the different states during the execution of the algorithm. Each agent calculates the cost of the sub-tree it is a root of in the BFS-tree and passes it to its parents. The root agent calculates the complete cost of each state and if it is found to be the best state found so far, propagates its index to the rest of the agents. Each agent \( A_i \) is required to hold its assignments in the last \( 2 \times d_i \) steps where \( d_i \) is the length of the route of parents in the BFS-tree from \( A_i \) to the root agent and is bounded by the height of the BFS-tree (\( h \)).

Next, we describe in details the actions agents perform in the ALS\_DCOP framework regardless of the algorithm in use. In each step of the algorithm an agent collects from its children in the BFS-tree the calculation of the cost of the sub-tree of which they are the root of. When it receives the costs for a step \( j \) from all its children, it adds its own cost for the state in step \( j \) and sends the result to its parent. When the root agent receives the calculations of the cost of step \( j \) from all its children, it calculates the global state cost. If it is better than the best state found so far, in the next step it will inform all its children that the state in step \( j \) is the best state found so far. Agents which are informed of the new best step store their assignment in that step as the best assignment and pass the information about the best index to their children in the next
DSA in ALS_DCOP
1. height ← height in the BFS-tree
2. dist ← distance from root
3. best ← null
4. best_index ← null
5. current_step ← 0
6. if (root)
   7. best_cost ← ∞
8. value_current ← ChooseRandomValue()
9. while (current_step < (m + dist + height))
   10. send value and cost_i to parent
   11. send value to non tree neighbors
   12. send value and best_index to children
   13. collect neighbors’ values
   14. cost_i ← CalculateStepCost(current_step − height)
   15. if (root)
   16. if (cost_i < best_cost)
   17. best_cost ← cost_i
   18. best ← value_i
   19. best_index ← i
   20. if (message from parent includes a new best_index j)
   21. best ← value_j
   22. best_index ← j
   23. if (ReplacementDecision())
   24. select and assign the next value
   25. delete value_i (current_step − (2 * dist))
   26. delete cost of step (current_step − height)
   27. current_step + +
28. for (1 to dist + height)
   29. receive message from parent
   30. if (message from parent includes a new best_index j)
   31. best ← value_j
   32. best_index ← j
   33. send best_index to children

Fig. 3. DSA in the ALS_DCOP framework.

The code for DSA in the ALS_DCOP framework is presented in Figure 3. The structure of the framework is homogeneous for all algorithms with a distributed synchronous local search general structure (such as DSA and DBA). It is interleaved in the algorithm execution as follows:

---

2 We assume the existence of a BFS tree when the algorithm begins.
1. In the initialization phase, besides choosing a random value for the variable, agents initialize the parameters which are used by the framework. The root initializes an extra integer variable to hold the cost of the best step (lines 1-7 in Figure 3).

2. In order to get the best out of \( m \) steps of the algorithm, \( m + h \) steps are performed (notice that for each agent the sum of height and dist is equal to \( h \) which is the height of the global BFS-tree). This is required so all the information needed for the root agent to calculate the cost of the \( m \) steps will reach it (line 9 in Figure 3).

3. After values are exchanged, each agent calculates the cost of the state according to its height. An agent with height \( h_i \) calculates the cost of the state in which its sub-tree was in \( h_i \) steps ago. The root agent checks if the cost it calculated is smaller than the best cost found so far and if so saves its information. All other agents check if the best index received from their parent is new. If so they save the information (index and assignment) of the step with the corresponding index (lines 16-22 in Figure 3).

4. Before the step is over, the agent deletes the information that has become redundant. This includes the information on the cost which it passed to its parent on this step and the assignment of the step which its index should have been received on this step in case it was found to be better than previous steps by the root agent (lines 25,26 in Figure 3).

5. On the next step, the value message an agent sends to its parent will include the cost calculation it had performed in this step and the messages to its children will include the index of the best step it knows of.

6. When the termination condition of the algorithm is met, the agents perform additional \( h \) steps in which only the best index is propagated down the tree. This way, if the last step cost calculated by the root agent is found to be best, its propagation to all agents is completed. Furthermore, by performing these steps, the possibility that different agents hold best assignments which belong to different steps is prevented (lines 28-33 in Figure 3).

An example of the performance of the ALS\_DCOP framework is presented in Figures 4 to 6. To keep the example simple, we only demonstrate the accumulation of the cost of a single step and the propagation of its index once it is found as the best so far. The figures do not show that while the costs of step \( i \) are being accumulated, costs and indexes of adjacent steps are also being passed by agents in the BFS-tree.

A DCOP in which the dashed lines connect neighbors in the constraint network and the arrows represent the BFS-tree arcs (each arrow is from parent to child) is presented on the left hand side of Figure 4. The costs in the figure are the private costs calculated for each agent to its state at step \( i \). In the next step, all the leaf agents in the BFS-tree (agents 3, 4 and 5) send their costs to their parents in the tree and the parents add their private costs to the costs they receive from their children. The resulting state is depicted on the right hand side of Figure 4 in which agent 2 added the costs for step \( i \) it received from its children agents 4 and 5 to its own cost of step \( i \) and got a cost of 8 for step \( i \). Agent 1 received the cost of agent 3 and added it to its own cost but it still did not receive the cost for step \( i \) from agent 2. At the next step, agent 1 receives the cost of step \( i \) from agent 2 and can calculate the total cost of step \( i \) (see the left hand side of Figure 5). Since it is smaller than the best cost achieved so far, agent 1 updates the
Fig. 4. On the left - Private costs of agents in step $i$. On the right - Calculations of the cost of step $i$ at step $i + 1$.

Fig. 5. On the left - Calculations of the cost of step $i$ at step $i + 2$. On the right - Propagation of the new best index, step $i + 3$.

Fig. 6. Propagation of the new best index, step $i + 4$.

new best cost to be 15 and in the next step sends a notification about a new best index in its messages to its children in the $BFS$-tree (see the right hand side of Figure 5). In the next step (Figure 6), the rest of the agents receive the notification that they should preserve the assignment they held in step $i$. Since the height of the $BFS$-tree is 2, the process of accumulating the cost of step $i$ by the root agent and the propagation of the information that it was found to be the best step took 4 steps.
5 Exploration Heuristics

The standard use of local search algorithms for DisCSPs and DCOPs prior to the proposal of the ALS_DCOP framework included running the algorithm for some number of iterations ($M$) and reporting the complete assignment (solution) held by the agents after the $M$th iteration. This use of the algorithm favored exploitive algorithms such as MGM and DSA over explorative algorithms like DBA [22].

The ALS_DCOP framework allows the selection of the best solution traversed by the algorithm and thus can encourage the use of explorative methods. We propose a number of heuristics which implement different approaches towards exploration:

- The first heuristic type we propose combines two exploration strategies which were found successful in previous studies. The first is a periodic increase in the level of exploration for a small number of iterations. This approach was found successful for the DCOP model proposed for mobile sensing agent teams DCOP_MST [25].
  The second is periodic restarts which in the case of local search methods result in a selection of a random assignment periodically. The random restart strategy is commonly used in constraint programming methods, e.g., [20]. We used the DSA-C version of DSA as the basic platform on which we incorporated the heuristics which incorporated these two strategies of search. In our basic version of DSA-C an agent replaces its assignment in a 0.4 probability if its best alternative value assignment does not increase the cost of its current assignment. The following two heuristics were combined with DSA-C:

  1. **DSA-C-PPIRA1**: PPIRA stands for *Periodic Probability Increase and Random Assignments*. Every 15 iterations we increased the probability of replacing an assignment to 0.8 for 5 iterations. Random selections in which each agent selects a random assignment were performed every 35 iterations.

  2. **DSA-C-PPIRA2**: Every 8 iterations we increased the probability of replacing an assignment to 0.9 for 5 iterations. Random selections in which each agent selects a random assignment were performed every 50 iterations.

- The second exploration approach we implemented formulates a dependency between the probability for replacing an assignment and the potential for improvement that this replacement offers. Such a dependency was suggested for the DSA algorithm in the DSA-B version. I DSA-B agents would not replace assignments if the number of violated constraints was zero. This method is compatible for distributed CSP problems where the aim is to satisfy all constraints. However, it is not applicable for DCOPs for which there is always some endured cost for a pair of assignments of constrained agents. Thus we propose the following heuristic that implements this approach: The heuristic is denoted by DSA-SDP where SDP stands for *Slope Dependent Probability*: If there is an improving alternative the probability for replacing the assignment is calculated as follows:

  \[
  p = 0.65 + \min(0.25, \frac{\text{current cost} - \text{new cost}}{\text{current cost}})
  \]

  If the alternative is not improving the current cost the probability is calculated as follows:
\[ p = \begin{cases} 
\frac{\text{current cost} - \text{new cost}}{\text{current cost}} > 1, & 0 \\
\frac{\text{current cost} - \text{new cost}}{\text{current cost}} \leq 1, & \max(0.1, 0.4 - \frac{\text{current cost} - \text{new cost}}{\text{current cost}})
\end{cases} \]

In this case (that the best alternative is not improving) we change in probability \( p \) only every 40 iterations to allow the algorithm time to converge.

- The last heuristic implements the approach of random restarts in a more monitored way. Here we do not select a complete assignments randomly but rather have single agents select a random assignments when they detect an over exploitive situation. We use the ability of the DBA algorithm to detect quasi local optima states and have agents detecting such a situation, break out of them by selecting a random assignment. We call this algorithm DRB which stands for Distributed Random Breakout. Like in the DBA algorithm the quasi local optima detected by the algorithm are states in which the best alternative of an agent and all its neighbors are not improving the current state. The difference is in the method for escaping this state which is random in DRB in contrast to DBA where the structure of the problem is changed by adding weights to constraints.

6 Experimental Evaluation

In order to emphasize the impact of the ALS DCOP framework on distributed local search, a set of experiments that demonstrate the effect of the proposed framework when combined with intensive exploration methods is presented.

Three different types of problems were used in the experiments, random DCOPs, graph coloring, and meeting scheduling. These problem were selected to demonstrate the effectiveness of the framework and the proposed heuristics on uniform, structured and realistic problems.

The uniform problems we used in our experiments were minimization random DCOPs in which each agent holds a single variable. Each variable had ten values in its domain. The network of constraints in each of the experiments, was generated randomly by selecting the probability \( p_1 \) of a constraint among any pair of agents/variables. The cost of any pair of assignments of values to a constrained pair of variables was selected uniformly between 1 and 10. Such uniform random DCOPs with constraint networks of \( n \) variables, \( k \) values in each domain, a constraint density of \( p_1 \) and a bounded range of costs/utilities are commonly used in experimental evaluations of centralized and distributed algorithms for solving constraint optimization problems [4].

All our experiments on random DCOPs included 120 agents. Each data point represents an average of 50 runs of the algorithm solving different problems.

In our first experiment the probability for a constraint between two agents (density parameter \( p_1 \)) was set to 0.2. Figure 7 presents the cost of the state in each iteration of the different algorithms. On the left hand side the results presented are for existing local search algorithms for DCOPs. On the right, the results of local search algorithms combined with the exploration methods we propose in this paper are presented. It is quite clear that the trends in these two graphs are very different. On the left hand side we
Fig. 7. The cost in each iteration of local search algorithms when solving random DCOPs, $p_1 = 0.2$.

Fig. 8. Anytime cost in each iteration of the local search algorithms when solving random DCOPs, $p_1 = 0.2$.

have exploitive algorithms while on the right it is apparent that the algorithm perform intensive exploration.

Figure 8 presents the anytime results for all algorithms on the same random setup. The three exploration heuristics which we combine with DSA outperform the other algorithms. For the DRB algorithm, a larger number of iterations is required (approximately 300) to reduce the cost to a level which is lower than the cost of the solutions found by existing algorithms.

Similar results were obtained for much more dense DCOPs ($p_1 = 0.6$). The results for this setup are presented in Figure 9. While most heuristics perform quite similarly to the more sparse case, the DSA-C-PIRA2 heuristic has a huge improvement in its anytime performance after 200 iterations. It seems that the extreme explorative selection of parameters allows it to traverse states which no other algorithm explores on dense problems.

The second set of experiments were performed on graph coloring problems. Each graph coloring problem included 120 agents, and as before, each data point represents
Fig. 9. Anytime cost in each iteration of the local search algorithms when solving random DCOPs, $p_1 = 0.6$.

Fig. 10. Anytime cost in each iteration of the local search algorithms when solving graph coloring problems.

an average of 50 runs of the algorithm solving different problems. The number of colors in the problem (domain size) was 3 and the density parameter $p_1 = 0.05$.

Figure 10 presents the results of the algorithms when solving graph coloring problems. Once again the heuristics combined with DSA are most successful. However, the DSA-SDP method outperforms the others. It seems that the structure of the graph coloring problem is exploited best by this method. On the other hand, there is a large gap in between the results obtained by the different versions of DSA and MGM as well to the the results of the two versions of DBA we compared with.

The last set of experiments were performed on realistic Meeting Scheduling Problems (MSPs) [3, 9, 10]. The agents’ goal in a MSP is to schedule meetings among them. We designed the problem as a minimization problem, thus, agents set their preferences by giving lower costs to meetings which are more important to them. In addition, for every two meetings we selected randomly an arrival time that is required to get from
Anytime cost in each iteration of the local search algorithms when solving meeting scheduling problems.

the location of one meeting to the other. When the difference between the time-slots of the meetings was less than the arrival time their was a cost endured. The cost was the number of agents participating in these meetings. The setup in this experiment included 90 agents and 20 meetings. There were 20 available time-slots for each meeting. The arrival times between meetings were selected randomly between 6 and 10.

The results in Figure 11 indicate once again that on problems with structure DSA-SDP performs best. On the other hand, in contrast to the graph coloring problems, the DRB algorithm performs well and is competitive with some of the DSA versions.

7 Conclusions

The ALS_DCOP framework enhances synchronous local search algorithms for DCOPs with the anytime property. This property enables the use of intensive exploration search methods which were too risky to use before. The negligible cost in time, space and communication load of the framework makes it most attractive. However, existing local search algorithms were designed to converge to a high quality solution at the end of the run and avoided extreme exploration.

In this paper we proposed a set of methods which are substantially more explorative than former local search algorithms. Our experimental evaluation reveals that using these methods within the ALS_DCOP framework improves the quality of the reported solution. Our experimental study included random, structured and realistic problems. It is apparent from our results that explorative heuristics dominate exploitive heuristics on all problem structures. However, for different problem types different exploration heuristics had the advantage.

In future work we intend to investigate the relation between the problem’s structure and the heuristic type.

References

Abstract. In this paper we focus on solving DCOPs in computationally demanding scenarios. GDL optimally solves DCOPs, but requires exponentially large cost functions, being impractical in such settings. Function filtering is a technique that reduces the size of cost functions. We improve the effectiveness of function filtering to reduce the amount of resources required to optimally solve DCOPs. As a result, we enlarge the range of problems solvable by algorithms employing function filtering.

1 Introduction

Distributed constraint optimization (DCOP) is a model for representing multi-agent systems in which agents cooperate to optimize a global objective. There are several complete DCOP algorithms that guarantee global optimality such as ADOPT [7], DPOP [8], and its generalization GDL [1, 13]. Since DCOPs are NP-Hard [7], solving them requires either an exponential number of linear size messages (ADOPT), or a linear number of exponentially large messages (DPOP, GDL).

Nonetheless, some application domains are specially resource constrained. For instance, in wireless sensor networks, memory, bandwidth, and computation are severely limited [15]. As computational requirements grow, so does the relevance of resource constraints. Hence, in this paper we focus on computationally demanding scenarios. An approach in these domains is to drop optimality in favor of lower complexity approximate algorithms with weaker guarantees [10]. As an alternative, we aim at reducing resource usage while keeping optimality.

Function filtering [12] is a technique that reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution. Function filtering can be applied to GDL, as detailed in [3], where they present the so-called DIMCTEf algorithm. Provided a lower bound on the cost of the best extension of the tuple, and an upper bound on the cost of the optimal solution, a tuple is filtered out when its lower bound is greater than the upper bound. Thus, the amount of filtered tuples strongly depends on
the quality of both lower and upper bounds. The tighter the bounds, the larger the size reduction of cost functions.

In [9], the authors reduce DIMCTEf communication needs whilst keeping memory and computation needs. However, this alternative turns out to be infeasible in computationally demanding scenarios. The purpose of this paper is to improve the effectiveness of function filtering by tightening the lower and upper bounds in [3]. More effective filtering reduces the amount of resources required to optimally solve DCOPs. Empirically, we estimate a reduction of up to 72% on communication costs and up to 32% on computational costs. Furthermore, we also obtain a significant memory reduction, allowing agents to solve up to 75% more problem instances given the same constraints. To summarize, we manage to increase the range of problems that can be solved optimally by algorithms employing function filtering.

The structure of the paper is as follows. Section 2 introduces DCOPs, and Section 3 outlines GDL with function filtering. Section 4 focuses on improving lower bounds, whereas Section 5 focuses on improving upper bounds. Finally, Section 6 draws some conclusions.

2 DCOP

A Distributed Constraint Optimization Problems (DCOP) is defined as a tuple $(X, D, C, A, \alpha)$, where:

- $X = \{x_1, \ldots, x_n\}$ is a set of $n$ variables.
- $D = \{D(x_1), \ldots, D(x_n)\}$ is a collection of domains, where $D(x_i)$ is the finite set of $x_i$’s possible values.
- $C$ is a set of cost functions. Each $f_S \in C$ is defined on the ordered set $S \subseteq X$, and specifies the cost of every combination of values of variables in $S$, namely $f_S : \prod_{x_j \in S} D(x_j) \mapsto \mathbb{R}^+$. $S$ is $f_S$’s scope, and $|S|$ is its arity.
- $A$ is a set of $p$ agents.
- $\alpha : X \rightarrow A$ maps each variable to some agent.

The objective of DCOP algorithms is to find the assignment of individual values to variables, such that the total (aggregated) cost over all cost functions is minimized. We make the common assumption that there are as many agents as variables, each agent controlling one variable, so from now on the terms variable and agent will be used interchangeably.

Next, we introduce concepts and operations that will be used throughout the paper.

A tuple $t_S$, with scope $S$, is an ordered set of values assigned to each variable in $S \subseteq X$. The cost of a complete tuple $t_X$, that assigns a value to each variable $x_i \in X$ is the addition of all individual cost functions evaluated on $t_X$. If a complete tuple’s cost is lower than the user-specified threshold, it is a solution. A minimum cost solution is optimal.

The projection $t_S[T]$ of a tuple $t_S$ to $T \subseteq S$ is a new tuple $t_T$, which only includes the values assigned to the variables in $T$. The combination of two cost
functions \( f_S \) and \( f_T \), written \( f_S \bowtie f_T \), is a new cost function \( f_U \) defined over their joint domain \( U = S \cup T \), s.t.:
\[
\forall t_U \quad (f_S \bowtie f_T)(t_U) = f_S(t_U[S]) + f_T(t_U[T])
\]
Combination is an associative and commutative operation.

Let \( F = \{ f_{T_1}, \ldots, f_{T_m} \} \) be a set of functions, the combination of \( F \), \( \bowtie F \), is the function resulting from the joint combination of every function in \( F \),
\[
\bowtie F = f_{T_1} \bowtie \ldots \bowtie f_{T_m}.
\]
Therefore, solving a DCOP means finding the tuple \( t_X \) that minimizes \( (\bowtie C)(t_X) \).

**Lower bounds to the cost of extensions of tuples**

Let \( V \subseteq X \) be a subset of the variables of the problem and \( t_V \) a tuple that assigns values to each of the variables in \( V \). An extension of \( t_V \) to \( X \) is a tuple that keeps the assignments of \( t_V \) and assigns new values to the variables in \( X\setminus V \). If the cost of each possible extension of \( t_V \) is larger than or equal to \( LB \), we say that \( LB \) is a lower bound of the cost of the best extension of tuple \( t_V \).

Likewise, a function \( f_T \) is a lower bound of function \( f_S \), noted \( f_T \leq f_S \), iff \( T \subseteq S \), and
\[
\forall t_S \quad f_T(t_S[T]) \leq f_S(t_S).
\]
A function \( g_V \) is a lower bound of a set of functions \( F \) if it is a lower bound of its combination \( \bowtie F \). In particular, a function \( g_V \) is a lower bound of a problem if it is a lower bound of the combination of its set of cost functions \( C \). Namely, if for each tuple \( t_V \), \( g_V(t_V) \) is a lower bound of the cost of the best extension of \( t_V \) to the complete problem.

The tightest lower bound is provided by the min-marginal. The min-marginal \( f_S[T] \) of a cost function \( f_S \) over \( T \subseteq S \) is a new cost function \( f_T \), which assigns to each tuple \( t_T \) the minimum cost among all the extensions of \( t_T \) to \( S \). Formally,
\[
\forall t_T \quad f_T(t_T[T]) = \min_{t_S \text{ extension of } t_T} f_S(t_S[T]).
\]
Similarly, the min-marginal of \( F \) over \( V \) is the min-marginal of the combination of all the functions in \( F \), that is \( (\bowtie F)[V] \).

Given a set of functions \( F \), the time to compute the min-marginal of \( F \) over \( V \) is bounded by \( \mathcal{O}(d_T^{|T|}) \), where \( T = \bigcup_{i=1}^m T_i \), and \( d_T \) is the maximum domain among the variables in \( T \). In some scenarios, this can be overdemanding. For that reason we introduce a less costly way of computing a lower bound of a set of functions. Specifically, we define \( \bowtie V F \), the combination of \( F \) under \( V \) as the result of combining the min-marginals of each of its functions over \( V \). That is,
\[
\bowtie V F = f_{T_1}[T_1 \cap V] \bowtie \ldots \bowtie f_{T_m}[T_m \cap V].
\]
\( \bowtie V F \) is a lower bound of \( F \) and can be assessed in \( \mathcal{O}(\max_{i=1}^m d_{T_i}^{|T_i|} \cdot d_V^{|V|}) \) time, which can be way smaller than \( \mathcal{O}(d_T^{|T|}) \).
Several algorithms can optimally solve DCOPs. In particular, we consider the GDL algorithm [1], following the Action-GDL description [13]. GDL works over a special structure named junction tree (JT), also known as joint tree or cluster tree. Action-GDL runs two phases: (1) costs are sent from the leaves up to the root; (2) optimal assignments are decided and communicated down the tree.

3 GDL

In Action-GDL, each node in the JT is controlled by an agent and has a set of variables $V_i$ and a subset of the cost functions of the problem $C_i$ (representing its own stake at the problem). We define the neighbors of $i$, $\text{neigh}(i)$, as the set of nodes linked to $i$ in the JT. Figure 1 shows agent $i$ and $j$ linked by an edge in a JT. Observe that removing the edge connecting $i$ and $j$ splits the JT into two different connected components, which we call subproblems. Formally, we say that the $i$-subproblem involves every cost function in the component containing $i$ after the edge is removed. Subproblems $i$ and $j$ are coupled by a set of variables they share and must agree upon, namely their separator $S_{ij} = V_i \cap V_j$.

Action-GDL determines the optimal solution using the following scheme. First, each leaf node starts by sending a lower bound of the costs of its subproblem to its parent. When a node has received the messages from all its children, it combines them to produce the lower bound of its own subproblem, and sends it to its parent. Once the root has received messages from all its children, it decides the best assignment (minimum cost tuple) for its variables. Then, it broadcasts this assignment to its children, who assess their best assignments and send them down the tree.

A main drawback of Action-GDL is the exponential size of the cost functions exchanged. To reduce communication, these messages can be approximated by means of lower arity functions [5], at the expense of losing optimality. The quality of the solution is expected to grow as the arity increases. We can take advantage of that to build an algorithm that successively finds better solutions by increasing the arity of approximations. As introduced in [9] there are two different schemes to approximate the functions exchanged in a junction tree: top-down and bottom-up. Top-down schemes compute the whole function to approximate,
and subsequently split it into smaller arity functions. Conversely, bottom-up schemes avoid computing the function to approximate, directly assessing smaller arity functions by combining subsets of functions. Since our interest lies in computationally intensive scenarios, the cost of assessing the function to approximate can be prohibitive, and hence in the following we restrict to bottom-up schemes.

3.1 GDL with function filtering

In this section we detail how function filtering can be applied to cope with the exponential growth of cost functions in GDL. Function filtering [12] is a technique that reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution. In order to do that, each agent $i$ intending to send a cost function $f_U$ to agent $j$ needs: (1) a lower bound $lb_U(t_U)$ on the value of each tuple $t_U$; and (2) an upper bound $UB$ on the value of the optimal solution. Provided that, we say that the agent filters $f_U$ with $lb_U$ and $UB$ when it removes those tuples $t_U$ such that $lb_U(t_U) > UB$ (i.e. the ones that cannot be extended into an optimal solution), and sends the remaining ones. Obviously, the amount of filtered tuples will strongly depend on the quality of bounds $lb_U$ and $UB$. The tighter the bounds, the larger the size reduction of cost functions.

Agents are unable to build a reasonably tight lower bound with the information they have in Action-GDL. Nevertheless, computing the lower bound of the best-cost extension of a tuple has been studied for the centralized case in MCTE($r$) [4], where $r$ is the highest arity of functions that can be computed. Its extension to the distributed setting is DMCTE($r$) [3], where functions of arity greater than $r$ cannot be exchanged. DMCTE($r$) works on a JT in three phases. The first one, cost propagation, builds a summary of each subproblem of the JT. The second one, solution propagation, computes a candidate solution. The third one, bound propagation, assesses the cost of the candidate solution.

During the cost propagation phase of DMCTE($r$) agents exchange approximate cost functions (one or several functions of arity up to $r$), first bottom-up and then top-down the JT. After this, each agent $i$ has: (1) a set of functions $C_i$, containing its stake at the problem; and (2) for each neighbor $j$ a set of functions $G_{j \rightarrow i}$. $G_{j \rightarrow i}$ stands for a summary of the $j$-subproblem, namely a lower bound on the cost of each tuple in that subproblem. Observe that agent $i$ can assess the cost of an assignment by adding its own costs and the costs of its neighbors’ subproblems. Likewise, the agent can assess a lower bound for the costs of a tuple in the complete problem by combining its own cost functions with those received from its neighbors. Formally,

$$lb_{V_i}(t_{V_i}) = (\otimes F)(t_{V_i}) \quad (1)$$

where $F = C_i \cup \bigcup_{j \in \text{neigh}(i)} G_{j \rightarrow i}$. However, the lower bound assessed in Equation (1) requires $O(d_{V_i}^{|V_i|})$ time, where $V_i$ are the agent’s variables. Hence, it cannot be computed in computationally demanding scenarios.
As an alternative, [3] proposes that each agent $i$ willing to send function $g_U$, where $U \subseteq V_i$, assesses a lower bound $lb_U(t_U)$ by adding: (1) a lower bound on the cost of the best extension of tuple $t_U$ in the $j$-subproblem to (2) the cost that $g_U$ assigns to $t_U$. Formally,

$$lb_U(t_U) = (g_{j \rightarrow i}^U \bowtie g_U)(t_U),$$

where $g_{i \rightarrow j}^U = \bowtie_U G_{i \rightarrow j}^U$. Henceforth we shall refer to this lower bound as **one-sided lower bound**. Since, in this case, the complexity of combining under $U$ is $O(|U|)$, this turns out to be much cheaper than using Equation (1).

The solution propagation phase of DMCTE($r$) works as Action-GDL’s. However, the solution obtained after this phase is not necessarily optimal. This is because DMCTE($r$) operates with approximate cost functions. Thereafter, DMCTE($r$) introduces a third phase to assess the cost of the candidate solution. Such cost is aggregated from the leaves to the root and then communicated down the tree. Observe that the cost of the candidate solution is an upper bound on the cost of the optimal solution.

Computing lower and upper bounds allows us to apply function filtering. Note that, when DMCTE($r$) runs with increasing $r$, the cost of the best solution found so far constitutes an upper bound $UB$. Furthermore, at each iteration, each agent $i$ willing to send a set of cost functions to agent $j$ can filter each one of them separately. Thus, for each function: first, it uses Equation (2) to assess a lower bound from the last message received from $j$; and then, it filters the function with the lower bound and $UB$. The resulting algorithm is known as DIMCTEf [3].

## 4 Two-sided filtering

Next, we aim at tightening the one-sided lower bound described above. Consider that agent $i$ has already received $G_{i \rightarrow j^*}$ from agent $j$. After that, it intends to send a set of functions $G_{i \rightarrow j}$, summarising the cost information in the $i$-subproblem, to agent $j$. Since no cost function appears in both the $i$-subproblem and the $j$-subproblem, we can assess a lower bound for the complete problem by adding a lower bound of each of them. Notice that the one-sided lower bound in Equation (2) already assesses the summary of the costs of the $j$-subproblem from $G_{j^* \rightarrow i}$. Likewise, we can assess the summary of the costs of the $i$-subproblem from $G_{i \rightarrow j^*}$. Therefore, we can employ the cost summaries of both subproblems to obtain a tighter bound.

Formally, when sending cost function $g_U \in G_{i \rightarrow j}$, we compute the lower bound of tuple $t_U$ as:

$$lb_U(t_U) = (g_{j \rightarrow i}^U \bowtie g_{i \rightarrow j})(t_U)$$

where

$- g_{i \rightarrow j}^U = \bowtie_U G_{i \rightarrow j}$ is a lower bound on the contribution of the $i$-subproblem.
\( g_{ji}^{j-i} = \bigcup G^{j-i} \) is a lower bound on the contribution of the \( j \)-subproblem.

Observe that there is no double counting of costs because no cost function appears in both the \( i \)-subproblem and the \( j \)-subproblem. Henceforth, we will refer to the lower bound in Equation (3) as two-sided lower bound. The name stems from the symmetrical use of both subproblems. Hereafter, two-sided filtering refers to filtering employing the two-sided lower bound.

\[
\begin{array}{c|c|c|c|c}
\text{Given} & \text{x y} & g_{xy}^{j-i} & \text{x y} & f_{xy} & \text{x z} & f_{xz} \\
\hline
\text{a a} & 3 & \text{a a} & 5 & \text{a a} & 4 \\
\text{a b} & 4 & \text{a b} & 2 & \text{a b} & 3 \\
\text{b b} & 3 & \text{b b} & 8 & \text{b a} & 5 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{x y} & \text{f}_{xy} \bowtie f_{xz}[x] = g_{xy}^{j-i} & \text{One-sided} & \text{Two-sided} & g_{xy}^{j-i} \bowtie g_{xy}^{j-i} \\
\hline
\text{a a} & 5 & 3 & 8 & 5 + 3 & 8 + 3 X \\
\text{a b} & 2 & 3 & 5 & 2 + 4 & 5 + 4 \\
\text{b a} & 8 & 2 & 10 & 8 + 3 X & 10 + 3 X \\
\text{b b} & 6 & 2 & 8 & 6 + 3 & 8 + 3 X \\
\end{array}
\]

Fig. 2: Example of one-sided vs. two-sided filtering. Tuples ticked off (\( X \)) are the ones being filtered out.

For instance, consider that agent \( i \) has received a set of functions \( G^{j-i} \), which combined under \( \{x, y\} \) produces the function \( g_{xy}^{j-i} \) shown in Figure 2. Furthermore, agent \( i \) knows that the cost of the optimal solution is smaller than or equal to 10 (\( UB = 10 \)). Now, it wants to send functions \( G^{j-i} = \{f_{xy}, f_{xz}\} \) (in Figure 2) to agent \( j \). Consider that it starts by sending function \( f_{xy} \). Agent \( i \) can calculate the one-sided lower bound using Equation (2), filtering out tuple \( (x=b, y=a) \) as shown in Figure 2. Alternatively, the agent can compute the two-sided lower bound using Equation (3), by assessing the lower bound on the contribution of its own subproblem, namely \( g_{xy}^{j-i} \bowtie g_{xy}^{j-i} = f_{xy} \bowtie f_{xz}[x] \). Figure 2 shows that two-sided filtering performs better, keeping only the tuple \( (x=a, y=b) \) as feasible.

4.1 Empirical evaluation

In this section we empirically compare the performance of DIMCTEf when using one-sided filtering and two-sided filtering. For each experiment, we track the amount of communication used by the algorithm (i.e., the total number of bytes) along with the total amount of serial computation (i.e., the number of non-concurrent constraint checks). Moreover, we performed signed rank tests [14] on all results to ensure that differences between methods are statistically significant (\( \alpha = 0.01 \)).
It is well-known that a JT’s treewidth is the most important indicator of problem hardness for GDL-based algorithms. Hence, we segmented our experiments according to this parameter, and ensured that all algorithms use the very same JT when solving the same problem instance. As mentioned before, we are specifically interested in hard problems. Thus, our first experiment used the hardest dataset in the DCOP repository, namely the meeting scheduling dataset [6]. We obtained very similar results for both one-sided and two-sided filtering, with very small gains for two-sided filtering.

As a consequence, we decided to design new datasets harder than those typically used in the DCOP literature. We characterized each scenario by three parameters: number of variables, variables’ domain size, and treewidth. For each scenario, we generated 100 problems by: (1) randomly drawing problem structures following an Erdős-Rényi $G(n, p)$ model [2]; (2) selecting those structures having the treewidth requested for the scenario; and (3) randomly drawing costs from a $\mathcal{N}(0, 1)$ distribution.

Fig. 3: Experimental results of one-sided filtering against two-sided filtering.
First, we ran an experiment to evaluate the savings as the treewidth increases. We generated scenarios with 100 variables of domain 8, and treewidths ranging from 6 to 9. Figure 3a shows that two-sided filtering reduces, with respect to one-sided filtering, the amount of communication required by a median of 26% for the easier problems (treewidth 6). It achieves even better results for the harder problems (52% for the set with treewidth 9).

Next, we designed an experiment to measure the trend of both filtering styles as the variables’ domain sizes increase. Thus, we generated scenarios with 100 variables, treewidth 9 and domain sizes ranging from 2 to 8. Once again, two-sided filtering achieves significant communication savings for all the experiment’s problems. Further, as the domain increases, so do the savings with respect to one-sided filtering: starting with a narrow 8% reduction for the binary variables set, and reaching a 52% reduction for the toughest scenario (domain size 8).

Furthermore, note that in all but the easiest experiments (the ones with variables’ domains 2 to 4), two-sided filtering performs up to 15% less non-concurrent constraint checks. Because function size is the main limiting factor of GDL-based algorithms, this suggests that two-sided filtering can solve problems that are too hard for one-sided filtering. Therefore, we re-ran the hardest set of problems (domain size 8, treewidth 9), but now limiting the maximum amount of memory available for each agent. Figure 3c shows that, indeed, two-sided filtering solves as much as 67% more problems than one-sided filtering.

5 Improving upper bounds

As mentioned above, the tighter the bounds, the larger the size reduction of cost functions. Previous work has focused on trying to improve the lower bounds sent during the cost propagation phase [11, 9]. In contrast, there is no work addressing the improvement of upper bounds (UB), despite also playing an important role in function filtering. Recall that the cost of any candidate solution is an upper bound on the cost of the optimal solution. Therefore, exploring multiple candidate solutions at the same time, instead of a single one, is expected to lead to better bounds. In this section, we present different approaches to propagate multiple candidate solutions. Then, we experimentally show that the benefits of providing the filtering process with better upper bounds can outweigh the cost of calculating them.

5.1 Centralized exploration

The simplest approach to propagating multiple assignments is to perform the very same procedure as DMCTE(r) does, but with multiple assignments instead. This is, the root node begins by choosing the best m assignments for its variables, and subsequently sends them to its children. Thereafter, each child extends each assignment by choosing its best cost extension (according to its knowledge), and relays them to its own children. The solution propagation phase terminates once each leaf node has received (and extended) its assignments.
Then, agents need to calculate the cost of each solution. With this aim, there is a third phase where: (1) the cost of each assignment is aggregated up the tree; and (2) the best assignment and its cost (the new global UB) are sent down the tree. Firstly, each leaf node $i$ evaluates the cost of each assignment in its problem’s stake $C_i$, and sends the resulting costs to its parent. Subsequently, once a parent node $j$ receives the costs of each assignment from its children, it aggregates them with the costs in its own problem stake $C_j$. Thereafter, it sends the resulting costs up the tree. After the root has received and aggregated the costs from all its children, it decides the best assignment. Finally, the root sends the best assignment along with its cost down the tree.

The main advantage of this method lays in its simplicity. However, its main drawback is that it offers limited exploration capabilities because: (1) it cannot propagate more than $k$ candidate solutions, where $k$ stands for all possible assignments for the root’s variables; and (2) when a node finds several good extensions for a candidate solution, it is enforced to choose only one of them. For instance, say that an agent receives assignment $(x=a)$ from its parent, and has to choose a value for variable $y$. According to its knowledge, extension $(x=a,y=a)$ costs 1, and so does extension $(x=a,y=b)$. Because centralized exploration forces the agent to extend each received assignment exactly once, extension $(x=a,y=b)$ must be discarded. This restriction implies that the root is the only node able to explore new candidate solutions, whereas other nodes simply exploit them.

5.2 Distributed exploration

To overcome the limited exploration capabilities of centralized exploration, we need mechanisms allowing any node to explore new assignments.

First, we assume that the number of assignments considered by each agent is bounded by $m$. Then, we enforce nodes to extend each received assignment at least once. However, we allow each agent to extend any assignment multiple times. Provided a node receives a set of assignments $A$, it needs to decide the number of new assignments to explore $n_e$, which cannot exceed $n_{\text{max}} = m - |A|$. With this aim, we propose that an agent employs one out of the following strategies:

**Greedy.** An agent extends as many assignments as possible, namely $n_e = n_{\text{max}}$.

**Stochastic.** An agent chooses the number of assignments to extend $n_e$ from a binomial distribution $B(n_{\text{max}}, p)$, where $p$ is the ratio of assignments to extend. Intuitively, higher $p$ values will favor exploitation, whereas lower $p$ values will favor exploration.

It might happen that the number of extensions requested $n_e$ is larger than the number of possible extensions. In that case the agent will communicate every possible extension. The process to calculate the cost of each solution is analogous to the one described for centralized exploration. The difference lies in the aggregation of costs up the tree. Since an agent may extend a parent’s assignment multiple times, it will send up the best cost out of the different extensions.
5.3 Empirical evaluation

To assess the performance of GDL with two-sided function filtering and the tighter upper bounds obtained by propagating multiple solutions, we ran experiments in the same scenarios we used in Section 4.1. Specifically, we assessed the communication and computation savings obtained by: (1) centralized exploration; (2) greedy distributed exploration; and (3) stochastic distributed exploration. Regarding the stochastic case, we empirically observed that different exploration ratios (different values for $p$), do not lead to very significant differences when filtering. Here we set $p=0.1$ because it provided slightly better results.

Note that, the harder the problem, the cheaper to propagate multiple solutions with respect to the cost propagation phase. Hence, we ran our experiments with different numbers of propagated solutions, and found that propagating 1024 solutions yielded the best results overall. This is, propagating less than 1024 solu-
tions slightly decreased the computation and communication used when solving the easier problems, but significantly increased when solving the harder ones. Likewise, propagating more solutions led to no additional savings on harder problems, while increasing costs on easier ones.

Figure 4a shows the evolution of the median results as the treewidth increases. On the one hand, centralized exploration achieves between 1 and 4% extra communication savings on top of two-sided filtering. On the other hand, both greedy and stochastic exploration outperform centralized exploration, consistently saving a median 20% communication cost, for a grand total of up to 72% savings when compared to the state-of-the-art one-sided filtering. Figure 4b displays very similar trends as variables’ domain grows. Centralized exploration provides a low reduction in communication, whereas greedy and stochastic exploration strategies obtain up to 24% extra savings with respect to two-sided filtering.

Finally, it is important to note that both greedy and stochastic exploration further reduce the number of non-concurrent constraint checks by as much as 24%. Furthermore, the reduction of computational effort goes up to 32% once multiple solutions propagation strategies are combined with two-sided filtering. Figure 4c reveals the effect of this reduction on the number of problems that can be solved when nodes have limited memory. Specifically, using two-sided filtering with distributed exploration helps solve up to 75% more problems than one-sided filtering.

6 Conclusions

Function filtering [12] is a technique that reduces the size of cost functions by filtering out tuples that are found unfeasible to be extended into an optimal solution. Function filtering can be readily applied to GDL, as detailed in [3]. This paper improves the effectiveness of state-of-the-art function filtering by providing techniques to assess tighter lower and upper bounds. Such improvements lead to significant reductions in the amount of resources required to optimally solve DCOPs. Thus, we can reduce up to 72% on communication costs and up to 32% on computational costs. Furthermore, we also obtain a significant memory reduction, allowing agents to solve up to 75% more problem instances given the same constraints. To summarize, we increased the range of problems that can be solved optimally by algorithms employing function filtering.

References


DisChoco 2: A Platform for Distributed Constraint Reasoning

Mohamed Wahbi¹,², Redouane Ezzahir³, Christian Bessiere¹ and El Houssine Bouyakhf²

¹ LIRMM/CNRS, University Montpellier II, France
² LIMIARF/FSR, University Mohammed V Agdal, Morocco
³ ENSA Agadir, University Ibn Zohr, Morocco
{wahbi,bessiere}@lirmm.fr, red.ezzahir@gmail.com and bouyakhf@fsr.ac.ma

Abstract. Distributed constraint reasoning is a powerful concept to model and solve naturally distributed constraint satisfaction/optimization problems. However, there are very few open-source tools dedicated to solve such problems: DisChoco, DCOPolis and FRODO. A distributed constraint reasoning platform must have some important features: It should be reliable and modular in order to be easy to personalize and extend, be independent of the communication system, allow the simulation of agents on a single virtual machine, make it easy for deployment on a real distributed framework, and allow agents with a local complex problems. This paper presents DisChoco 2.0, a complete redesign of the DisChoco platform that guarantees these features and that can deal both with distributed constraint satisfaction problems and with distributed constraint optimization problems.

1 Introduction

Distributed Constraint Reasoning (DCR) is a framework for solving various problems arising in Distributed Artificial Intelligence. In DCR, a problem is expressed as a Distributed Constraint Network (DCN). A DCN is composed of a group of autonomous agents where each agent has control of some elements of information about the problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. Agents try to find a local solution (locally consistent assignment) and communicate it with other agents using a DCR protocol to check its consistency against constraints with variables owned by other agents [1,2].

A DCN offers an elegant way for modelling many everyday combinatorial problems that are distributed by nature (e.g., distributed resource allocation [3], distributed meeting scheduling [4], sensor networks [5]). Several algorithms for solving this kind of problems have been developed. Asynchronous Backtracking (ABT [6], ABT-Family [7]), Asynchronous Forward Checking (AFC) [8] and Nogood-based Asynchronous Forward-Checking (AFC-ng) [9] were developed to solve Distributed Constraint Satisfaction Problems (DisCSP). Asynchronous Distributed constraints Optimization (Adopt) [10], Asynchronous Forward-Bounding (AFB) [11], Asynchronous
Branch-and-Bound (Adopt-BnB) [12] and Dynamic backtracking for distributed constraint optimization (DyBop) [13] were developed to solve Distributed Constraint Optimization Problems (DCOP).

Programming DCR algorithms is a difficult task because the programmer must explicitly juggle between many very different concerns, including centralized programming, parallel programming, asynchronous and concurrent management of distributed structures and others. In addition, there are very few open-source tools for solving DCR problems: DisChoco, DCOPolis [14] and FRODO [15]. Researchers in DCR are concerned with developing new algorithms, and comparing their performance with existing algorithms. Open-source platforms are essential tools to integrate and test new ideas without having the burden to reimplement from scratch an ad-hoc solver. For this reason a DCR platform should have the following features:

– be reliable and modular, so it is easy to personalize and extend;
– be independent from the communication system;
– allow the simulation of multi-agent systems on a single machine;
– make it easy to implement a real distributed framework;
– allow the design of agents with local constraint networks.

In this paper we present DisChoco 2.0,\(^1\) a completely redesigned platform that guarantees the features above. DisChoco 2.0 allows to represent both DisCSPs and DCOPs, as opposed to other platforms. DisChoco 2.0 is not a distributed version of the centralized solver Choco, but it implements a model to solve DCN with local complex problems (i.e., several variables per agent) by using Choco\(^2\) as local solver to each agent. DisChoco 2.0 is an open source Java library which aims at implementing DCR algorithms from an abstract model of agent (already implemented in DisChoco). A single implementation of a DCR algorithm can run as simulation on a single machine, or on a network of machines that are connected via the Internet or via a wireless ad-hoc network, or even on mobile phones compatible with J2ME.

This paper is organized as follows. Section 2 presents the global architecture of DisChoco 2.0. In Section 3, we show how a user can define her problem and solve it using the DisChoco 2.0 platform. Section 4 shows the different benchmarks available in DisChoco and how researchers in the DCR field can use them for evaluating algorithms performance. We conclude the paper in Section 5.

## 2 Architecture

In order to reduce the time of development and therefore the cost of the design we choose a components approach allowing pre-developed components to be reused. This components approach is based on two principles:

– Each component is developed independently;
– An application is an assemblage of particular components.

\(^1\)http://www.lirmm.fr/coconut/dischoco/
\(^2\)http://choco.emn.fr/
Figure 1 shows the general structure of DisChoco kernel. It shows a modular architecture with a clear separation between the modules used, which makes the platform easily maintainable and extensible.

![Figure 1: Architecture of DisChoco kernel](image1)

The kernel of DisChoco consists of an abstract model of an agent and several components namely the communicator, messages handlers, constraints handler, the Agent View (AgentView), a Master who controls the global search (i.e., send messages to launch and to stop the search, etc.) and a communication interface.

**Communication System**

Thanks to independence between the kernel of DisChoco and the communication system that will be used (Figure 2), DisChoco enables both: the simulation on one machine and the full deployment on a real network. This is done independently of the type of network, which can be a traditional wired network or an ad-hoc wireless network.

![Figure 2: Independence between the kernel of DisChoco and the communication system](image2)
Instead of rewriting a new system of communication between DisChoco agents we adopted the component approach. Thus a communication component pre-developed can be used as a communication system if it satisfies a criterion of tolerance to failure. This allows us to use only the identifiers of agents (IDs) to achieve communication between agents. Thus when agent $A_i$ wants to send a message to the agent $A_j$, it only attaches its ID ($i$) and the ID ($j$) of the recipient. It is the communication interface that will deal with mapping between the IDs and IP addresses of agents (we assume that an agent identifier is unique).

In the case of a simulation on a single Java Virtual Machine agents are simulated by Java threads. Communication among agents is done using an Asynchronous Message Delay Simulator (MailerAMDS) [16,17]. MailerAMDS is a simulator that models the asynchronous delays of messages. Then, agents IDs are sufficient for communication. In the case of a network of Java Virtual Machines, we have used SACI $^3$ (Simple Agent Communication Infrastructure) as communication system. The validity of this choice has not yet been validated by an in depth analysis. Future work will be devoted to testing a set of communication systems on different types of networks.

**Event Management**

DisChoco performs constraint propagation via events on variables and events on constraints, as in Choco. These events are generated by changes on variables, and managing them is one of the main tasks of a constraint solver. In a distributed system there are some other events that must be exploited. These events correspond to a reception of a message, changing the state of an agent (wait, idle and stop) or to changes on the AgentView.

The AgentView of a DisChoco agent consists of external variables (copy of other agents variables). Whenever an event occurs on one of these external variables, some external constraints can be awakened and so added to the queue of constraints that will be propagated. Using a queue of constraints to be propagated allows to only process constraints concerned by changes on the AgentView instead of browsing the list of all constraints. To this end, the DisChoco user can use methods offered by the constraints handler (**ConstraintsHandler**). Detecting the termination of a distributed algorithm is not a trivial task. It strongly depends on statements of agents. To make the implementation of a termination detection algorithm easy, we introduced in the DisChoco platform a mechanism that generates events for changes on the statements of an agent during its execution. A module for detecting termination is implemented under each agent as a listener of events on statements changes. When the agent state changes, the termination detector receives the event, recognizes the type of the new state and executes methods corresponding to termination detection.

The events corresponding to an incoming message are managed in DisChoco in a manner different from the standard method. Each agent has a Boolean object that is set to false as long as the inbox of the agent is empty. When a message has arrived to the inbox, the agent is notified by the change of this Boolean object to true. The agent can

---

use methods available in the communicator module to dispatch the received message to its corresponding handler.

**Observers in layers**

DisChoco provides a Java interface (AgentObserver) that allows the user to track operations of a DCR algorithm during its execution. This interface defines two main functions: whenSendMessage and whenReceivedMessage. The class AbstractAgent provides a list of observers and functions to add one or several observers. Thus, when we want to implement an application using DisChoco, we can use AgentObserver to develop a specific observer. This model is shown in Figure 3(a).

When developing new algorithms, an important task is to compare their performance to other existing algorithms. There are several metrics for measuring performance of DCR algorithms: non-concurrent constraint checks (#ncccs [18]), equivalent non-concurrent constraint checks (#encccs [19]), number of exchanged messages (#msg [20]), degree of privacy loss [21], etc. DisChoco simply uses AgentObserver to implement these metrics as shown in Figure 3(b). The user can enable metrics when she needs them or disable some or all these metrics. The user can develop her specific metric or her methods for collecting statistics by implementing AgentObserver.

![Figure 3: Layer model for observers.](image)

**3 Using DisChoco 2.0**

Figure 4 presents a definition of a distributed problem named (Hello DisChoco) using the Java code. In this problem there are 3 agents $A = \{A_1, A_2, A_3\}$ where each agent controls exactly one variable. The domain of $A_1$ and $A_2$ contains two values $D_1 = D_2 = \{1, 2\}$ and that of $A_3$ contains one value $D_3 = \{2\}$. There are two constraints of difference: the first constraint is between $A_1$ and $A_2$ and the second one is between $A_2$ and $A_3$. After defining our problem we can configure our solver. Thus, the problem can be solved using a specified implemented protocol (ABT for example).

For DisChoco inputs we choose to use a XML format called XDisCSP derived from the famous format XCSP 2.1. Figure 5 shows an example of representation of the problem defined above in the XDisCSP format. Each variable has a unique ID, which is the

---

4 http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html
AbstractMaster master = Protocols.getMaster(Protocols.ABT);
DisProblem disCSP = new DisProblem("Hello DisChoco", master);
SimpleAgent[] agents = new SimpleAgent[3];
IntVar[] variables = new IntVar[3];
// Make agents
agents[0] = (SimpleAgent) disCSP.makeAgent("A1", "");
agents[1] = (SimpleAgent) disCSP.makeAgent("A2", "");
agents[2] = (SimpleAgent) disCSP.makeAgent("A3", "");
// Make one single variable for each agent
variables[0] = agents[0].makeInternalVar(new int[] {1, 2}); // x1
variables[1] = agents[1].makeInternalVar(new int[] {1, 2}); // x2
// Make two constraints, we must to create external var on each agent
// But each agent must know its constraints
// x1!=x2
agents[0].neqY(agents[0].makeExternalVar(variables[1]));
agents[1].neqY(agents[1].makeExternalVar(variables[1]));
// x2!=x3
agents[1].neqY(agents[1].makeExternalVar(variables[2]));
agents[2].neqY(agents[2].makeExternalVar(variables[1]));
// Make a simulator to resolve the problem
DisCPSolver solver = new DisCPSolverDisChoco(disCSP);
solver.setCentralizedAO(new LexicographicAO());
solver.addNCCCMetric();
solver.solve();
System.out.println("Statistics :");
system.out.println(solver.getStatistics());

<instance name="Hello DisChoco" model="Simple" maxConstraintArity="2" format="XDisCSP 1.0">
<agents nbAgents="3">
<agent name="A1" id="1" description="Agent 1 ">
<agent name="A2" id="2" description="Agent 2 ">
<agent name="A3" id="3" description="Agent 3 ">
</agents>
<domains nbDomains="2">
<domain name="A1" nbValues="2">1 2</domain>
<domain name="A2" nbValues="2">1 2</domain>
<variables nbVariables="3">
<variable name="x1" id="1" domain="A1" description="Variable x_1 ">
<variable name="x2" id="2" domain="A2" description="Variable x_2 ">
<variable name="x3" id="3" domain="A3" description="Variable x_3 ">
</variables>
<predicates nbPredicates="1">
<predicate name="PO">
<parameters>x int y</parameters>
<expression>
(functional) x = (x,y)</expression>
</predicate>
</predicates>
<constraints nbConstraints="2">
<constraint name="C1" min="1" max="2" scope="X1.0 X2.0" reference="PO">
<parameters>X1.0 X2.0</parameters>
</constraint>
<constraint name="C2" min="2" max="2" scope="X2.0 X3.0" reference="PO">
<parameters>X2.0 X3.0</parameters>
</constraint>
</constraints>
</instance>

---

Fig. 4: Definition of a distributed problem using Java code.

Fig. 5: Definition of the Hello DisChoco problem in XDisCSP 1.0 format.
concatenation of the ID of its owner agent and index of the variable in the agent. This is necessary when defining constraints (scope of constraints). For constraints, we used two types of constraints: TKC for Totally Known Constraint and PKC for Partially Known Constraint [21]. Constraints can be defined in extension or as a Boolean function. Different types of constraints are predefined: equal to $eq(x, y)$, different from $ne(x, y)$, greater than or equal $ge(x, y)$, greater than $gt(x, y)$, less than or equal $le(x, y)$, less than $lt(x, y)$, etc.

According to this format we can model DisCSPs and DCOPs. Once a distributed constraint network problem is expressed in the $XDisCSP$ format, we can solve it using one of the protocols developed on the platform. The algorithms currently implemented in DisChoco 2.0 are: ABT [6,7], ABT-Hyb [22], ABT-dac [23], AFC [8], AFC-ng [9], DBA [24] and DisFC [21] in the class of DisCSPs with simple agents. In the class of DisCSPs where agents have local complex problems, ABT-cf [25] was implemented. For DCOPs, the algorithms that are implemented in DisChoco 2.0 are: Adopt [10], BnB-Adopt [12] and AFB [11]. For solving a problem, we can use a simple command line:

```java
java -cp dischoco.jar dischoco.simulation.Run protocol problem.xml
```

The Graphical User Interface (GUI) of DisChoco allows to visualise the constraint graph. Hence, the user can analyse the structure of the problem to be solved. This also helps to debug the algorithms. An example of the visualisation is shown in Figure 6.

![Visualisation of the structure of the distributed constraint graph.](image)

Fig. 6: Visualisation of the structure of the distributed constraint graph.

## 4 Experimentations

In addition to its good properties (reliable and modular), DisChoco provides several other facilities, especially for performing experimentation. The first facility is in the
generation of benchmark problems. DisChoco offers a library of generators for distributed constraint satisfaction/optimization problems (e.g., random binary DisCSPs using model B, random binary DisCSPs with complex local problems, distributed graph coloring, distributed meeting scheduling, sensor networks, distributed N-queens, etc.). These generators allow the user to test her algorithms on various types of problems ranging from purely random problems to real world problems.

DisChoco is equipped with a GUI for manipulating all above generators. A screenshot of the GUI of DisChoco shows various generators implemented on DisChoco (Figure 7). Once the instances have been generated, a XML configuration file is created to collect the instances. The generated instances are organized in a specific manner for each kind of problems generator in a directory indicated by the user. The configuration file can also contain details related to the configuration of the communicator and the list of algorithms to be compared. It will be used for launching experiments. After all these configurations have been set, the user can launch the experiments either on the GUI mode or on the command mode.

Fig. 7: A screenshot of the graphical user interface showing generators in DisChoco

DisChoco is also equipped with a complete manager of results. The user does not have to worry about organizing and plotting results. All this is offered by DisChoco that automatically generates gnuplot plots of the requested measures. The user can also handle all results and compare algorithms using the GUI of DisChoco. Figure 8 shows an example of plot generated from experimentations on some algorithms implemented in DisChoco.

5 Conclusion

In this work, we have presented the new version 2.0 of the DisChoco platform for solving DCR problems. This version contains several interesting features: it is reliable and
modular, it is easy to personalize and to extend, it is independent from the communication system and allows a deployment in a real distributed system as well as the simulation on a single Java Virtual Machine. As future work, we aim at enhancing the platform by the implementation of other DCR algorithms and to enrich the graphical user interface to make it easier to use for researchers from the DCR field. Another direction of improvement is to allow DisChoco to support other types of constraints that match as much as possible the needs of real applications. The modularity of DisChoco will allow us to look for other types of system communication. Finally, for a complete validation, it would be interesting to test DisChoco on a real distributed system.

References