Explorative Anytime Local Search for Distributed Constraint Optimization

Roie Zivan, Steven Okamoto and Hilla Peled
Department of Industrial Engineering and Management,
Ben-Gurion University of the Negev,
Beer-Sheva, Israel
{*zivanr,okamotos,hillapel*}*@bgu.ac.il*

Abstract: Distributed Constraint Optimization Problems (DCOPs) are an elegant model for representing and solving many realistic combinatorial problems that are distributed by nature. DCOPs are NP-hard and therefore many recent studies consider incomplete algorithms for solving them. Distributed local search algorithms, in which agents in the system hold value assignments to their variables and iteratively make decisions on whether to replace them, 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 that 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 that enhances distributed local search algorithms for DCOPs with the *anytime* property is proposed. The proposed framework makes use of a BFS-tree in order to accumulate the costs of the system's state during the algorithm's iterative performance and to propagate the detection of a new best state when it is found. The proposed framework does not require additional network load. Agents are required to hold a small (linear) additional space (beside the requirements of the algorithm in use).

We further propose a set of increased exploration heuristics that exploit the proposed anytime framework. These exploration methods implement different approaches towards exploration. Our empirical study considers various scenarios including random, realistic, and structured problems. It reveals the advantage of the use of the proposed heuristics in the anytime framework over state-of-the-art local search algorithms.[1]

Key Words: Distributed Constraint Optimization, Incomplete search, Exploration Heuristics.

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 Sys-

[1]This paper is an extension of [1]. Besides an extended description and examples, it proposes new exploration heuristics that exploit the anytime framework and an intensive empirical study that reveals the advantages in using the proposed framework.

tems and has generated significant interest from researchers [2, 3, 4, 5, 6, 7, 8, 9]. 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 [4, 10].

There is a wide scope in the motivation for research on DCOPs, since they can be used to model many everyday combinatorial problems that are distributed by nature. Some examples are the *Nurse Shift assignment problem* [11, 12], the *Sensor Network Tracking problem* [7], and the *Log-Based Reconciliation problem* [13].

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

A number of studies on DCOPs presented complete algorithms [4, 5, 14]. However, since DCOPs are NP-hard, there has been growing interest in the last few years in local (incomplete) DCOP algorithms [15, 7, 1, 16, 17]. Although local search does not guarantee that the obtained solution is optimal, it can be applied to large problems and is compatible with real-time applications.

The general design of most state-of-the-art local search algorithms for DCOPs is synchronous [18, 7, 19, 20] (a notable exception was presented in [21]). In each step (or iteration) of the algorithm an agent sends its assignment to all its neighbors in the constraint graph and receives the assignments 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) [15], 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) [7].

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 that balance between exploration and exploitation are used to improve the current assignment of the algorithm [22, 23, 24]. An important feature of most centralized 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 remain the same or increase if more steps of the algorithm are performed [25]. This property cannot be guaranteed as easily in a distributed environment where agents are only aware of the cost of their own assignment (and maybe that of their neighbors too), but no one actually knows when a good global solution is obtained.

In [7], DSA and DBA are evaluated solving sensor network DCOPs. Apparently these algorithms perform well on this application even without a pure anytime property. The algorithms were compared by evaluating the state held by agents at the end of the run. However, Zhang et al. [7] do not offer a way to report the best state explored by the algorithms as proposed in this study. This limits the chances of local search algorithms implementing an exploring heuristic to be successful.

In order to implement anytime local search algorithms that follow the same syn-

chronous structure of DSA and DBA for distributed optimization problems, the global result of every synchronous step must be calculated and the best solution must be stored. A trivial approach would be to centralize in every step the costs calculated by all agents to a single agent. This agent would then inform the other agents each time a solution that improves the results on all previous solutions is obtained. However, this method has drawbacks both in the increase in the number of messages and in the violation of privacy caused from the need to inform a single agent (not necessarily a neighbor of all agents) of the quality of all other agents' states in each step of the algorithm.

The present paper proposes a general framework for enhancing local search algorithms for DCOPs that follow the general synchronous structure with the anytime property. In the proposed framework the quality of each state is accumulated via a spanning tree of the constraint graph. Agents receive information about the quality of the recent states of the algorithm from their $children$ in the spanning 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 of the system's state in each step and propagates down the tree the index of the step in which the system was in the most successful state. When the search is terminated, all agents hold the assignment of the best state according to the global goal function.

The proposed framework can be combined with any synchronous incomplete algorithm such as MGM, DSA, DBA, or Max-Sum. The combination allows any such algorithm to report the best solution it traversed during its run, i.e. it makes it an anytime algorithm.

In order to produce the best state out of m steps, the algorithm must run $m + 2h$ synchronous steps where h is the height of the tree used. Since the only requirement of the tree is that it is a spanning tree on the constraint graph, i.e., that it maintains a parent route from every agent to the root agent, the tree can be a BFS-tree and its height h is expected to be small (in the worst case h equals the number of agents n). Our experimental study reveals that starting from very low density parameters, the height of the BFS-tree is indeed very small (logarithmic in the number of agents). Previous studies on distributed systems have used BFS trees, e.g., for maintaining shortest paths in a communication network [26]. However, to the best of our knowledge we are the first to use it for aggregating local state information in order to make a decision on the global best solution.

The proposed framework does not require agents to send any messages in addition to the messages sent by the original algorithm. The additional space requirement for each agent is $O(h)$.

We study the potential of the proposed framework by proposing a set of exploration methods (heuristics) that exploit the anytime property by introducing extreme exploration to exploitive algorithms. We present an extensive empirical evaluation of the proposed methods on three different benchmarks for DCOPs. The proposed methods find solutions 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: Related work is presented in Section 2. Section 3 describes the distributed constraint optimization problem (DCOP). State-of-the-art local search algorithms for solving DCOPs are presented in Section 4.

Section 5 presents ALS_DCOP, the proposed anytime local search framework for DCOPs. In Section 6 we present the theoretical properties of the proposed framework. Section 7 proposes innovative exploration methods for distributed local search. Section 8 presents an experimental study that evaluates the proposed explorative algorithms when combined with the proposed anytime framework in comparison with existing local search algorithms. Section 9 presents a discussion of the experimental results and Section 10 presents our conclusions.

2. Related Work

A number of complete algorithms were proposed in the last decade for solving DCOPs. The simplest algorithm among these was the *Synchronous Branch and Bound* (*SynchB&B*) algorithm [27], which is a distributed version of the well-known centralized Branch and Bound algorithm. Another algorithm that uses a Branch and Bound scheme is *Asynchronous Forward Bounding* (*AFB*) [14], in which agents perform sequential value assignments that are propagated for bound checking and early detection of a need to backtrack. A number of complete algorithms use a pseudo-tree, which is derived from the structure of the constraints network, in order to improve the process of acquiring a solution. *ADOPT* and *BnB-ADOPT* [4, 28] are two such asynchronous search algorithms in which assignments are passed down the pseudo-tree. Agents compute upper and lower bounds for possible assignments and send costs, which are eventually accumulated by the root agent, up to their parents in the pseudo-tree. Another algorithm that exploits a pseudo-tree is *DPOP* [5]. In DPOP, each agent receives from the agents that are its children in the pseudo-tree all the combinations of partial solutions in their sub-tree and their corresponding costs. The agent calculates and generates all the possible partial solutions, which include the partial solutions it received from its children and its own assignments, and sends the resulting combinations up the pseudo-tree. Once the root agent receives all the information from its children, it identifies the value assignment that is a part of the optimal solution and propagates it down the pseudo-tree to the rest of the agents, allowing them to find their own value assignment and thus, produce the optimal solution. The DPOP algorithm is a GDL algorithm [29], i.e., it stems from the general distributive law that allows accumulation of costs and utility calculation operations performed by different agents in a distributed system for a mutual decision on the optimal solution. A number of recent studies investigate this paradigm and propose improvements to the DPOP algorithm [30, 31]. A very different approach was implemented in the *OptAPO* algorithm [3, 32] in which agents that are in conflict choose a mediator to whom they transfer their data and which solves the problem. This algorithm benefits when the underlying constraint graph includes independent sections that can be solved concurrently.

All of the algorithms mentioned above are complete. While this is an advantage in the sense that they guarantee to report the optimal solution, this is also a drawback since DCOPs are NP-hard; thus, in order to validate that an acquired solution is optimal they must traverse the entire search space in the worst case. This drawback limits the use of these algorithms to relatively small problems.

Some of the complete algorithms mentioned above use a tree structure and, like in the framework we propose in this paper, aggregate information to the root agent in

order to allow it to make a global decision on the optimal solution. However, our work is the first to suggest such an aggregation that can apply to any synchronous incomplete algorithm, even if the algorithm itself does not use a tree structure. Moreover, the tree structure used in complete algorithms must be a pseudo-tree while the proposed framework can use any spanning tree. This allows us to make use of a BFS tree which evidently has a much smaller (logarithmic) height.

One common approach towards incomplete methods for solving DCOPs is *distributed local search*. The general design of most local search algorithms for DCOPs is synchronous [18, 7, 20]. In each step of the algorithm an agent sends its assignment to all its neighbors in the constraint network and receives the assignments of all its neighbors. In Section 4 we present in detail two leading algorithms from this class of algorithms – the *Distributed Stochastic Algorithm* (*DSA*) [7] and the *Distributed Breakout Algorithm* (*DBA*) algorithm [18]. These algorithms were selected because many other algorithms such as MGM and DisPeL [33] are very similar to them and can be considered their descendants. In addition, the most successful exploration heuristics we propose in this paper are variations of these algorithms.

In [15, 34], a different approach towards local search was proposed. In these studies, completely exploitive algorithms are used to converge to local optima solutions, which are guaranteed to be within a predefined distance from the global optimal solution. The approximation level is dependent on a parameter k, which defines the size of coalitions that agents can form. These k-size coalitions transfer the problem data to a single agent, which performs a complete search procedure in order to find the best assignment for all agents within the k-size coalition. As a result, the algorithm converges to a state that is *k-optimal* [34], i.e., no better state exists if k agents or fewer change their assignments. While this approach guarantees that the outcome of the algorithm is *k-optimal*, the algorithms proposed to date that guarantee convergence to k-optimal solutions, such as MGM [15], include very limited exploration if at all (in contrast to algorithms that include intensive exploration that are within scope and aim of this paper). Recently, this approach of using monotonic local search algorithms in small local environments in order to produce quality guarantees on the solution was extended to environments dependent on the distance of nodes in the constraint network [21] and environments that are bounded both by distance and size [35].

A different approach towards incomplete distributed problem solving is implemented by the Max-Sum algorithm. Max-Sum [8] is a GDL algorithm that operates on a *factor graph*, which is a bipartite graph in which the nodes represent variables and constraints. Although Max-Sum is completely exploitive, it is not guaranteed to converge in problems with factor graphs that include cycles [8]. On such problems it performs implicit exploration. We present the details of the Max-Sum algorithm in Section 4 following the descriptions of DSA and DBA. In addition, we demonstrate in this paper the advantages of using Max-Sum within the proposed anytime framework.

A recent study has investigated the usefulness of increasing the level of exploration in DSA and DBA [36]. It proposed versions of DSA in which the probability for changing an assignment is higher than in standard DSA. These experiments compared with Distributed Simulated Annealing (DSAN), in which the probability to take an explorative step was dependent on a decreasing temperature. Beside standard DBA and MGM, this study also compared with the DisPeL algorithm, which is a penalty-driven

algorithm that increments unary constraints instead of binary constraints, as DBA does. We demonstrate that the explorative methods we propose in this paper outperform these alternative approaches when combined with the anytime framework.

Another recent study proposed intelligent functions for selecting the probability to replace assignments in DSA [37]. The proposed methods considered the potential for reduction in cost (the slope of improvement) and as a result gave agents with a higher probability to improve the global cost a higher probability to perform assignment replacements. The resulting versions of the algorithms were reported to converge faster and in implementations where only assignment replacements are exchanged, reduce significantly the number of messages exchanged. However, the quality of the obtained solutions was not improved when using the proposed functions. Nevertheless, the most successful method we propose in this paper combines this approach with the approach taken in DSAN that allows explorative assignment selection under some conditions and with monitored random restarts. This combination, when implemented within the proposed anytime framework, was found to dominate on all benchmarks we experimented with.

Both studies, [37] and [36], reported results on the number of messages sent between agents (triggered by assignment changes) because they considered an asynchronous version of DSA. In our work, we consider synchronous local search algorithms that perform in an asynchronous environment; thus, agents must exchange messages with all neighbors in every step of the algorithm.

3. Distributed Constraint Optimization

A *distributed constraint optimization problem* (*DCOP*) is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$. $\mathcal{A}$ 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_s. Each variable is held by a single agent (an agent may hold more than one variable). $\mathcal{D}$ is a set of domains D_1, D_2,...,D_s. Each domain D_i contains the finite set of values that can be assigned to variable X_i. $\mathcal{R}$ is a set of relations (constraints). Each constraint $C_i \in \mathcal{R}$ defines a non-negative *cost* for every possible value combination of a set of variables, and is of the form $C_i : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow \mathbb{Z}^+ \cup \{0\}$. Each agent A_j involved in constraint C_i holds a part of the constraint C_{i_j} so that $\sum_j C_{i_j} = C_i$. In this paper we will assume that for each pair of agents A_j and $A_{j'}$ involved in constraint C_{i_j}, $C_{i_{j'}}$ are equal, i.e., problems are symmetric. However, our proposed framework applies to the asymmetric DCOP framework as well [38]. A *binary constraint* involves exactly two variables and is of the form $C_i : D_{i_1} \times D_{i_2} \rightarrow \mathbb{Z}^+ \cup \{0\}$. A *binary DCOP* is a DCOP in which all constraints are binary. An *assignment* (or a label) is a pair of 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. *vars(PA)* is the set of all variables that appear in PA, $vars(PA) = \{X_i \mid \exists a \in D_i \wedge (X_i, a) \in PA\}$. A constraint $C_i \in \mathcal{R}$ of the form $C_i : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow \mathbb{Z}^+ \cup \{0\}$ is *applicable* to PA if $X_{i_1}, X_{i_2}, \ldots, X_{i_k} \in vars(PA)$. 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 ($vars(PA) = \mathcal{X}$). A *solution* is a full assignment returned by a DCOP algorithm.

In this paper, we assume each agent holds a single variable and use the term "agent" and "variable" interchangeably. We also assume that constraints are at most binary and that the delay in delivering a message is finite [4, 39]. Agents are aware only of their own local topology (i.e., only of their own neighbors in the constraint network and the constraints that they individually and privately hold).

We use the term *step* for a synchronous iteration of a local search algorithm. The *state* of an agent in each step includes its value assignment and the *local cost*, which is the sum of costs incurred according to constraints with the value assignments of its neighbors. The global state in each step includes the full assignment and the costs of all constraints violated by the full assignment.

4. Local Search for Distributed Constraint Problems

The general design of most state-of-the-art local search algorithms for Distributed Constraint Satisfaction Problems (DisCSPs) and Distributed Constraint Optimization Problems (DCOPs) is synchronous. In each step of the algorithm an agent sends its value assignment to all its neighbors in the constraint network and receives the value assignment of all its neighbors. Two of the most known algorithms that apply to this general framework are the Distributed Stochastic Algorithm (DSA) [7] and the Distributed Breakout Algorithm (DBA) [40, 7]. In our presentation of the algorithms we follow the recent versions of [7]. Notice that these algorithms were first designed for distributed constraint *satisfaction* problems in which a solution must not violate any constraint, but they can be applied without any adjustment to *Distributed Max-CSPs* (DisMaxCSPs) (where the optimal solution is the complete assignment with the smallest number of violated constraints), which is a specific type of DCOP. Thus, in our description we consider an improvement to be a decrease in the number of violated constraints (as in DisMaxCSPs).

4.1. Distributed Stochastic search Algorithm (DSA)

The basic idea of DSA is simple. After an initial step in which agents select a starting value for their variable (randomly according to [7]), 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 constraint graph and receives the value assignments of its neighbors.[2] After collecting the value 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 [7], if an agent in DSA cannot improve its current state by replacing its current value, it does not replace it. If it can improve (or keep the same cost, depending on the version used), it decides whether to replace the value using a stochastic strategy (see [7] for details on the possible strategies and the differences in the resulting performance). A sketch of DSA is presented in Figure 1. After a random value is assigned to the agent's

[2] 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.

DSA

1. $value \leftarrow$ 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

Figure 1: Standard DSA.

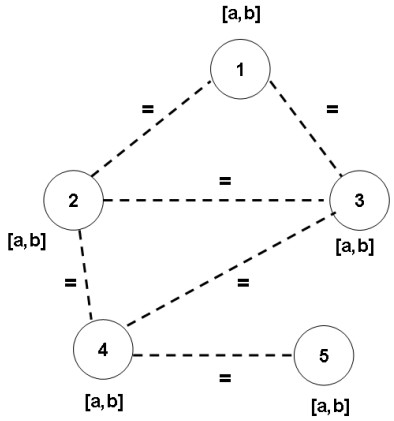

Figure 2: DCOP example.

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 value assignment; when the decision is positive it assigns a new value to its variable (lines 5,6). The version of the algorithm we used in our experiments was DSA-C in which the replacement decision is to replace its current value assignment with probability p if the alternative assignment does not deteriorate the local state (i.e., it either improves it or keeps the cost the same).

An example of a DCOP in which each constraint has the same cost (DisMaxCSP) 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 value. Assume agents 1, 3, and 5 selected a and agents 2 and 4 selected b. In the first step all agents can change their assignment without increasing local cost. Following a stochastic decision 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 if it does so, in the next step only agent 5 can replace its assignment. In the resulting state, all agents assign a to their variable and the algorithm converges.

4.2. Distributed Breakout Algorithm (DBA)

DBA manipulates the weights of the DCOP constraints. As in DSA, in every step each agent sends its current value assignment to its neighbors and collects their current value assignments. After receiving the value assignments of all its neighbors, the agent computes the maximal weighted cost reduction it can achieve by changing its value assignment and sends this proposed reduction to its neighbors. After collecting the proposed reductions from its neighbors, an agent changes its value assignment only if its proposed reduction is greater than that of its neighbors. If ties occur they are broken using the agents identifiers. When an agent detects a *quasi-local optimum*, i.e., neither it nor any of its neighbors offer a reduction of weighted cost, it increases the weights of its current constraint violations. The sketch of the DBA algorithm is depicted in Figure 3. After initializing the constraint weights to one and assigning a random value to its variable (lines 1,2), the agent enters the loop where, as in the DSA algorithm, each loop iteration is a step of the algorithm. After sending its value assignment to its neighbors and collecting their value assignments (lines 4,5), the agent calculates its best weight reduction and sends it to its neighbors (lines 6,7). After receiving the possible weight reduction of all of its neighbors the agent decides whether to replace its assignment and on a positive decision assigns its variable (lines 8-11). If it detects a quasi-local optimum it increases the weights of all its violated constraints (lines 13,14).

The Maximum Gain Messages (MGM) algorithm is a simplified version of DBA. It includes the same step structure as DBA, i.e. sending the maximal reduction in cost to neighbors and then sending the value assignment, but it does not include the manipulation of constraint weights to break out of quasi-local optima. In other words, MGM is completely monotonic. The code of MGM is similar to the code presented in Figure 3 excluding lines 12-14.

4.3. Max-Sum

A different incomplete approach for solving DCOPs is implemented by the Max-Sum algorithm. The Max-Sum algorithm [8] is a GDL algorithm [29] that operates on a *factor graph* [41], a bipartite graph in which both variables and constraints are represented by nodes.[3] Each node representing a variable of the original DCOP is connected to all function-nodes that represent constraints that the variable is involved in. Similarly, a function-node is connected to all variable-nodes that represent variables in the original DCOP that are included in the constraint it represents. Agents in Max-Sum perform the roles of different nodes in the factor graph. Variable-nodes and function-nodes are considered "agents" in Max-Sum, i.e., they can send messages, read messages and perform computation.

[3] We preserve the terminology of [8] and call constraint-representing nodes in the factor graph "function-nodes".

DBA

1. Set the local weights of constraints to one
2. $value \leftarrow$ ChooseRandomValue()
3. **while** (no termination condition is met)
4. send value to neighbors
5. collect neighbors' values
6. $WR \leftarrow$ BestPossibleWeightReduction()
7. Send WR to neighbors
8. Collect WRs from neighbors
9. **if** ($WR > 0$)
10. **if** ($WR > WRs$ of neighbors (ties broken using indices))
11. $value \leftarrow$ the value that gives WR
12. **else**
13. **if** (no neighbor can improve)
14. increase violated constraints' weights by one

Figure 3: Standard DBA.

Max-sum (node n)

1. $N_n \leftarrow$ all of n's neighboring nodes
2. **while** (no termination condition is met)
3. collect messages from N_n
4. **for each** $n' \in N_n$
5. **if** (n is a variable-node)
6. produce message $m_{n'}$ using messages from $N_n \setminus \{n'\}$
7. **if** (n is a function-node)
8. produce message $m_{n'}$ using constraint and messages from $N_n \setminus \{n'\}$
9. send $m_{n'}$ to n'

Figure 4: Standard Max-Sum.

Figure 4 presents a sketch of the Max-Sum algorithm. The code for variable-nodes and function-nodes is similar apart from the computation of the content of messages to be sent. For variable-nodes only data received from neighbors is considered. In messages sent by function-nodes the content is produced considering data received from neighbors and the original constraint represented by the function-node.

It remains to describe the content of messages sent by the factor graph nodes. A message sent from a variable-node x to a function-node f at iteration i, includes for each of the values $d \in D_x$ the sum of costs for this value it received from all function-node neighbors apart from f in iteration $i-1$. Formally, for value $d \in D_x$ the message will include:

$$\sum_{f' \in F_x, f' \neq f} cost(f'.d) - \alpha$$

where F_x is the set of function-node neighbors of variable x and $cost(f'.d)$ is the cost for value d included in the message received from f' in iteration $i-1$. α is a constant that is reduced from all costs included in the message (i.e., for each $d \in D_x$) in order to prevent the costs carried by messages throughout the algorithm from growing arbitrarily. Selecting α to be the average on all costs included in the message is a reasonable choice for this purpose [8, 42]. Notice that as long as the amount reduced from all costs is identical, the algorithm is not affected by this reduction since only the differences between the costs for the different values matter.

A message sent from a function-node f to a variable-node x in iteration i includes for each possible value $d \in D_x$ the minimal cost of any combination of assignments to the variables involved in f apart from x and the assignment of value d to variable x. Formally, the message from f to x includes for each value $d \in D_x$:

$$min_{ass_{-x}} cost(\langle x, d \rangle, ass_{-x})$$

where ass_{-x} is a possible combination of assignments to variables involved in f not including x. The cost of an assignment $a = (\langle x, d \rangle, ass_{-x})$ is:

$$f(a) + \sum_{x' \in X_f, x' \neq x} cost(x'.d')$$

where $f(a)$ is the original cost in the constraint represented by f for the assignment a, X_f is the set of variable-node neighbors of function-node f, and $cost(x'.d')$ is the cost that was received in the message sent from variable-node x' in iteration $i-1$, for the value d' that is assigned to x' in a.

In contrast to DSA, DBA and MGM, Max-Sum is not a search algorithm, i.e., the selection of value assignments to variables is not an inherent part of the algorithm and is not used to generate the messages in the algorithm. However, in every iteration an agent can select its value assignment. Each variable-node selects the value assignment that received the lowest sum of costs included in the messages that were received most recently from its neighboring function-nodes. Formally, for variable x we select the value $\hat{d} \in D_x$ as follows:

$$\hat{d} = min_{d \in D_x} \sum_{f \in F_x} cost(f.d)$$

Notice that the same information used by the variable-node to select the content of the messages it sends is used for selecting its value assignment.

5. Anytime Local Search Framework for DCOPs

Local search algorithms as presented in Section 4 combine exploration and exploitation properties in order to converge to local optima, and escape from them in order to explore other parts of the search space. When a centralized constraint optimization local search algorithm is performed, the quality of the states (i.e., full assignments) of the algorithm are completely known and therefore there is no difficulty in holding the best state that was explored. In a distributed constraint optimization problem, agents are only aware of their own private state (the violated constraints they are involved in and their costs) and, thus, a state that can seem to have high quality to a single agent might have low global quality and vice versa. In the case of Distributed Constraint *Satisfaction* Problems (DisCSPs), the problem is easier, since in the global optimal state all agents are in a private optimal state as well with no violated constraints. This is not the case in Distributed Constraint Optimization Problems (DCOPs) where the global optimal state can include a number of violated constraints. In this case the evaluation of the states according to the private preferences of an agent can be different from the evaluation of the states according to the global quality of states.

We note that the algorithms presented in Section 4 were originally designed for DisCSPs and have subsequently been adapted for solving DCOPs [7]. The problem is that even if the optimal state is reached, none of the agents will be aware of it. Since none of the agents are aware of the quality of the global state, the termination condition must be independent of the states' quality, for example, stopping after the algorithm performs a limited number of steps [7]. Moreover, the algorithm can only report the final state reached, not the state with the highest quality that it has explored.

We propose a framework, ALS_DCOP, that enhances DCOP local search algorithms with the *anytime* property. In the proposed framework, a spanning tree on the constraint graph is used, similar to ADOPT [4] and DPOP [5]. However, while ADOPT and DPOP require the use of a pseudo-tree in which every pair of constrained agents must be on the same branch in the tree, the only requirement in ALS_DCOP is that the tree is indeed a spanning tree on the constraint graph, i.e., every agent has a parent route to the root agent and all parents in the tree are neighbors of their children in the constraint graph. Thus, a *Breadth First Search* (*BFS*) tree on the constraint graph can be used.

A BFS-tree is a spanning tree (in this case it spans the constraints graph) that includes the shortest path from the root to each of the graph's nodes [43]. A BFS-tree can be generated efficiently using the following distributed procedure proposed in [26] for maintaining shortest routes in networks:

1. Each agent A_i holds a private variable δ_i which is initially set to ∞.
2. The root agent (e.g., agent with smallest index) sets δ_{root} to zero and sends to all its neighbors a message that includes this number (zero).
3. Each agent A_i that receives a message with a number j, smaller than δ_i, sets $\delta_i \leftarrow j + 1$, sets the agent it received the message from to be its parent in the tree, and sends messages with the new value of δ_i to each of its other neighbors.

4. After h steps, every agent knows its parent in the BFS-tree with height h.
5. Another h steps are needed so that every agent knows its height in the tree (similarly, leaves send height zero to parents, etc).
6. After $2h$ steps the BFS-tree is ready to be used by the algorithm.

The BFS-tree structure is used in order to accumulate the costs of agents' assignments 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 parent. The root agent calculates the complete cost of each state and if some state is found to be the best state so far, propagates its step index to the rest of the agents. Each agent A_i is required to hold its assignments in the last $h + d_i$ 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 h of the BFS-tree.

Next, we describe in detail the actions agents perform in the ALS_DCOP framework regardless of the algorithm in use. These actions are also described by the pseudocode in Figure 5. In the initialization phase, besides choosing a random value for the variable, agents initialize the parameters that are used by the framework (lines 1-10 in Figure 5). The root initializes an extra integer variable to hold the cost of the best step (line 10). In order to find the best state out of m steps of the algorithm, $m + h$ steps are performed (notice that for each agent the sum of h_i and d_i is equal to h, which is the height of the entire BFS-tree). This is required so that all the information needed for the root agent to calculate the cost of the m steps will reach it (line 11). 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 (line 15). 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 (line 17). When the root agent receives from all its children the costs of step j of the subtrees they are the roots of, it calculates the global state cost (lines 18–22). 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 that are informed of the new best step store their value assignment from that step as the best assignment and propagate the step number that was found to be best to their children in the next step (lines 23–25). After every synchronous step the agents can delete the information stored about any of the steps that were not found to be the best and are not of the last $h + d_i$ steps (lines 28–29 in Figure 5). When the local search algorithm is terminated, the agents must perform another h steps in which they do not replace their assignment to make sure that all the agents are aware of the same index of the best step (lines 31–36).

An example of the performance of the ALS_DCOP framework is presented in Figures 6 to 8. To keep the example simple, we only demonstrate the accumulation of the cost of a complete assignment in a single step and the propagation of its index once it is found to be the best so far. The figures do not show that while the costs of the agents' states in step i are being accumulated, costs and indices 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 side of Figure 6. The costs in the figure are the local (private) costs calculated for each agent in its state at step i. In the next step, all the leaf agents in the BFS-tree

ALS_DCOP

1. $h_i \leftarrow$ height in the BFS-tree
2. $d_i \leftarrow$ distance from root
3. $best \leftarrow null$
4. $best\_step \leftarrow null$
5. $step \leftarrow 0$
6. $cost_i \leftarrow 0$
7. $local\_cost^{step} \leftarrow 0$
8. $value_i^{step} \leftarrow$ selectValue()
9. **if** (root)
10. $\quad best\_cost \leftarrow \infty$
11. **while** $(step < (m + d_i + h_i))$
12. $\quad$ send message including $cost_i$ to parent
13. $\quad$ send messages to non tree neighbors
14. $\quad$ send message $best\_step$ to children
15. $\quad$ collect neighbors' values
16 $\quad local\_cost^{step} \leftarrow$ CalculateLocalCost()
17. $\quad cost_i \leftarrow$ CalculateStepCost($step - h_i$)
18. $\quad$ **if**(root)
19. $\quad\quad$ **if**($cost_i < best\_cost$)
20. $\quad\quad\quad best\_cost \leftarrow cost_i$
21. $\quad\quad\quad best \leftarrow value_i^{step}$
22. $\quad\quad\quad best\_step \leftarrow step$
23. $\quad$ **if** (message from parent includes a new $best\_step\ j$)
24. $\quad\quad best \leftarrow value_i^j$
25. $\quad\quad best\_step \leftarrow j$
26. $\quad$ **if** (ReplacementDecision())
27. $\quad\quad$ select and assign the next value
28. $\quad$ delete $value_i^{(step-(h+d_i))}$
29. $\quad$ delete $local\_cost^{(step-h_i)}$
30. $\quad step++$
31. **for** (1 to $d_i + h_i$)
32. $\quad$ receive message from parent
33. $\quad$ **if** (message from parent includes a new $best\_step\ j$)
34. $\quad\quad best \leftarrow value_i^j$
35. $\quad\quad best\_step \leftarrow j$
36. $\quad$ send $best\_step$ to children

Figure 5: ALS_ DCOP framework.

(agents 3, 4, and 5) send their local 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 side of Figure 6, in which agent 2 added the costs for step i it received from its children, agents 4 and 5, to its own local 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 local 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 for the complete assignment (see the left side of Figure 7). Since it is smaller than the best cost achieved so far, agent 1 updates the new best cost to be 15 and in the next step

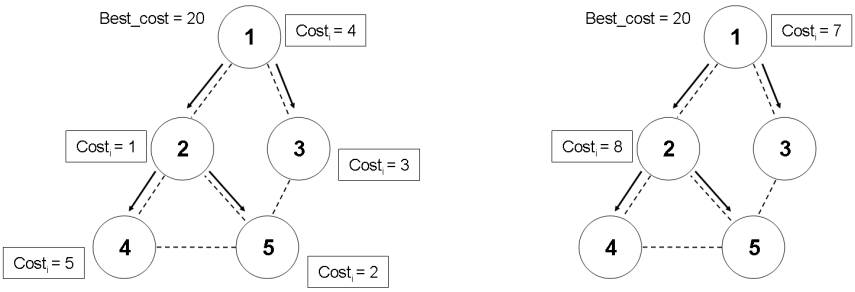

Figure 6: On the left - Private costs of agents in step i. On the right - Calculations of the cost of step i at step $i + 1$.

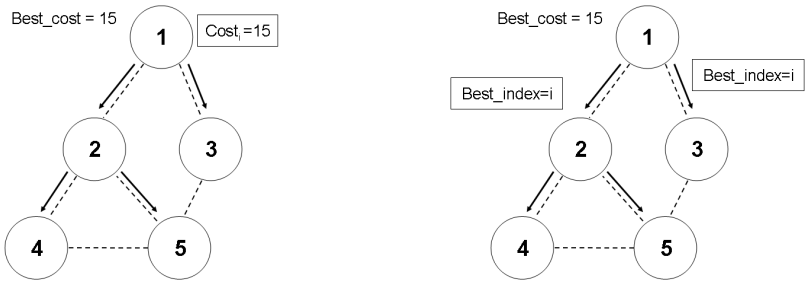

Figure 7: On the left - Calculations of the cost of step i at step $i + 2$. On the right - Propagation of the new best step, step $i + 3$.

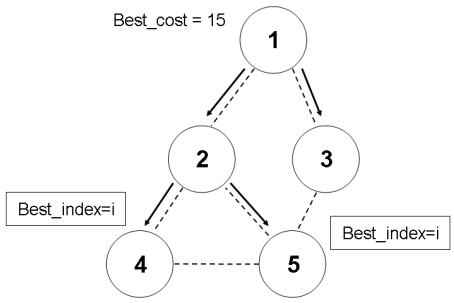

Figure 8: Propagation of the new best step, step $i + 4$.

sends a notification about a new best step in its messages to its children in the BFS-tree (see the right side of Figure 7). In the next step (Figure 8), 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 was found to be the best step took 4 steps.

6. Properties of the ALS_DCOP framework

ALS_DCOP is a framework for implementing local search algorithms for DCOPs. Regardless of the algorithm being used, the ALS_DCOP framework offers properties that ensure preservation of the algorithm's behavior.

6.1. Anytime property

The main goal of the ALS_DCOP framework is to enhance a distributed local search algorithm with the *anytime* property (i.e., that the cost of the solution held by the algorithm at the end of the run would monotonically decrease if the algorithm is allowed to run for additional steps [25]). In order to prove that ALS_DCOP is an *anytime* framework for distributed local search algorithms, we first prove the following lemma:

Lemma 1. *At the $i+h$ step, the root agent holds all the needed information for calculating the quality of the assignment held by the agents in step i.*

Proof: We prove by induction on the height of the tree h. If $h = 0$, the root is the only agent in the system and thus it holds all the relevant information. For $h > 0$, assume the lemma holds for every tree of height less than h. After $i + h - 1$ steps, according to the assumption, all the children of h hold all the information needed to compute the costs of the sub-tree that they are the roots of. In step $i + h$, the root agent will receive these costs and it will be able to add its own private cost for step i to the sum of the costs of all the other agents in the DCOP in step i. □

Next, we prove that at the end of the run (after $m + 2h$ steps) the best assignments held by all agents are value assignments that they held in the same step during the algorithm run.

Lemma 2. *When the algorithm terminates after $m + 2h$ steps, all the assignments of the best state held by all the agents in the system were the values they assigned to their variables in the same step (in other words the $best\_step$ for all agents is equal).*

Proof: According to Lemma 1, after $m + h$ steps of the algorithm, the root agent holds the index of the best step in the first m steps of the algorithm. In the final h steps only messages including the $best\_step$ are passed. Since no new $best\_step$ is found by the root in these steps and h steps are enough for all the agents in the BFS-tree to receive a new $best\_step$, then even if a $best\_step$ was found in the mth step of the algorithm, its propagation is completed. □

Now we are ready to state the main theorem:

Theorem 1. *The ALS_DCOP framework is* anytime*, i.e., when it runs for $m+2h$ steps it reports the best state among the first m steps.*

Proof: This follows directly from Lemmas 1 and 2. h steps after each step is performed, the root agent holds the information needed to evaluate its global quality (the aggregated cost) and can propagate its index to all other agents in case it is the best. Since we require that agents hold the value assignments of the last $h + d_i$ steps (where d_i is the length in a route of parents of an agent from the root and is at most h,

see Section 5), they can get the update on the index of the best step before they delete the relevant assignment and hold it until they receive another update. Thus, after $m+h$ steps, the root agent holds the index of the step with the best state among the first m steps and, according to Lemma 2, at the end of the algorithm run, all agents hold the value assignment of the step that was found by the root to be the best. If the algorithm is run for an additional k steps (i.e., the termination condition is met after $m + k + 2h$ steps), in the k steps performed after the first m steps either a better solution is found or the same solution that was found best in the first m steps is reported. □

6.2. Performance analysis

Distributed algorithms are commonly measured in terms of time for completion and network load. Time in a synchronous system is counted by the number of synchronous steps, and network load by the total number of messages sent by agents during the algorithm run [44]. When considering a local search algorithm for DCOPs, since it is not complete and the agents cannot determine that they are in an optimal state, the time for termination is predefined, i.e., we select in advance the number of steps that we intend to run the algorithm for. However, we note that in the proposed framework, in order to get the best state among m steps the algorithm needs to run for $m + 2h$ steps. However, the tree that is used by ALS_DCOP has different requirements than the pseudo-trees that are used by complete algorithms (such as ADOPT and DPOP). In a pseudo-tree that is used in complete algorithms, constrained agents must be on the same parent route (in every binary constraint, one of the constrained agents is an ancestor of the other). This requirement makes the pseudo-tree less effective when the constraint graph is dense. In contrast, the only requirement in ALS_DCOP is that every agent has a parent route to the root agent, allowing us to use a BFS-tree. Since a BFS-tree includes a shortest route from each node to the root, the height of the resulting tree (especially when the constraint graph is dense) is expected to be small. In our experimental study we demonstrate that starting from very low density parameters the height of the BFS-tree for randomly-generated problems is logarithmic in the number of agents.

In terms of network load, the ALS_DCOP framework does not require any additional messages. In standard local search algorithms (such as DSA and DBA), agents at each step send at least one message to each of their neighbors. The ALS_DCOP framework does not require more. The additional information that agents add to messages is constant (costs or best step indices).

In terms of space, an agent i is required to hold the value assignments of the last $h + d_i$ steps (again, d_i is the distance in tree edges of an agent from the root) and to hold the cost of its own state and probably of its neighbors in the last h_i steps (where h_i is the height of the agent in the BFS-tree). This results in a $O(h)$ additional space (linear in the number of agents the worst case, logarithmic in practice) requirement for each agent.

6.3. Privacy of ALS_DCOP

Local search algorithms require different amounts of information to be passed between the agents. In DSA, only the assignments are passed by agents to neighboring

agents, while in DBA, agents also pass their proposed reduction to the cost of the current state. ALS_DCOP requires, in addition, that each agent will pass the cost of a state in the sub-tree of which it is the root. As in other algorithms that use a tree (such as ADOPT and DPOP), the main problem with privacy in ALS_DCOP concerns the information passed by leaves in the tree to their parents [45].

When a non-leaf agent A_j passes the cost of its sub-tree to its parent, the parent does not know how many children A_j has and the contribution of each of these agents to the reported cost. On the other hand, when a leaf agent reports a cost, its parent knows that it is the cost of a single agent (the leaf itself). However, agents are not aware of the system's topology except for their own neighbors. So in fact, even though the parent of a leaf receives its cost in every step of the algorithm, the parent does not know how many neighbors its leaf child has in the constraint network and which constraints were violated; therefore the privacy violation is minor.

7. Exploration Heuristics

The standard use of local search algorithms for DisCSPs and DCOPs prior to the proposal of the ALS_DCOP framework includes running an algorithm for some number of steps (m) and reporting the complete assignment (solution) held by the agents after the $m'th$ step. This use of the algorithm favored exploitive algorithms such as MGM and DSA over explorative algorithms like DBA [7].

7.1. Exploration in Existing Incomplete Algorithms

MGM is quintessentially exploitive: it is a monotonic algorithm that implements distributed hill climbing and converges to a local minimum without exploration. Its successors MGM-2, DALO and K-opt [15, 21] are similarly monotonic.

DSA also implements distributed hill climbing and so is exploitive by design, but it is not monotonic because simultaneous updates by constrained agents may result in an increase in cost. With an appropriately selected probability for replacing an assignment by the agents (parameter p), the algorithm performs mostly exploitive (very limited exploration) and converges quickly [7]. However, as long as $p < 1$, DSA will converge to a local minimum in finite time, although it may take impractically long if p is set to be too great.

Agents in DSAN perform biased random walks by randomly choosing new value assignments and always adopting cost-decreasing changes while adopting cost-increasing changes with a probability that decreases over time. After a period of initial exploration, DSAN eventually converges to a local minimum.

Penalty-driven approaches like DBA and DisPeL modify the cost structure that the agents reason over by adding penalties when they can no longer reduce their local costs. DBA adds these penalties to constraints, while DisPeL adds them to values (i.e., unary constraints); the penalties eventually cause the agents to shift to different value assignments, thereby escaping the local minima. However, the subsequent local minima according to the modified cost structure that the algorithms converge to may not be local minima in the problem. This is a more radical form of exploration which may not converge.

In contrast to the existing local search algorithms whose levels of exploitation and exploration depend on their search strategies, Max-Sum performs a completely exploitive search strategy, always propagating the best costs and selecting the best assignment, but it is still not guaranteed to converge [8, 46]. When it fails to converge it acts exploitively but actually explores low quality states.

7.2. Innovative Exploration Methods

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 both algorithm-specific and algorithm-independent heuristics implementing different approaches towards exploration.

7.2.1. Algorithm-Specific Exploration Heuristics

The algorithm-specific heuristics that we propose extend DSA and DBA.

- The first heuristic type we propose combines two exploration strategies that were found to be successful in previous studies. The first is a periodic increase in the level of exploration for a small number of steps. This approach was found to be successful for the DCOP model proposed for mobile sensing agent teams DCOP_MST [19]. The second is periodically restarting, which in the case of local search methods results in a periodic selection of a random assignment. The random-restart strategy is commonly used in constraint programming methods, e.g., [47]. We incorporated this strategy with the DSA-C version of DSA. In DSA-C, an agent replaces its assignment with probability p (in our experiments we used $p = 0.4$) if its best alternative value assignment does not increase the cost of its current assignment. In the proposed heuristic, every k steps the probability of replacing an assignment is increased from p to p^* for k^* steps. In order to combine the restart strategy, every r steps agents select a random assignment. In our experiments we used two combinations of the parameters k, k^*, p, p^*, and r, which were found to be most successful. We refer to it as DSA-PPIRA. PPIRA stands for Periodic Probability Increase and Random Assignments.

- 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 [7]. In DSA-B, agents do not replace assignments if the number of violated constraints is zero. This method is compatible with distributed CSP problems where the aim is to satisfy all constraints. However, it is not applicable when solving DCOPs where there is always some incurred cost for a pair of assignments of constrained agents. Thus we propose the DSA-SDP heuristic, where SDP stands for *Slope Dependent Probability*: If there is an improving alternative, the probability of replacing the assignment is calculated as follows:

$$p = p_A + min\left(p_B, \frac{|current\_cost - new\_cost|}{current\_cost}\right)$$

where new_cost is the cost of the best alternative value assignment. Thus, the explorative value of parameter p is determined by the proportion of the new assignment in comparison to the current assignment. In order to maintain the explorative nature of the algorithm, p_A bounds p from below.[4]

If the alternative does not improve the current cost, the probability q is used for replacing the value assignment, calculated as follows:

$$q = \begin{cases} 0, & \frac{|current\_cost - new\_cost|}{current\_cost} > 1 \\ max\left(p_C, p_D - \frac{|current\_cost - new\_cost|}{current\_cost}\right), & \frac{|current\_cost - new\_cost|}{current\_cost} \leq 1 \end{cases}$$

Intuitively, although an improving assignment could not be found, we still want some means of exploration. However, we do not want to deteriorate the current result significantly. Thus, we consider the proportion between the new best possible utility (least worst) for an alternative assignment and the current utility. If this proportion is not too large (in the above formula the threshold was set to an addition of 100%) we avoid changing an assignment. Otherwise, we select a probability with respect to this proportion.[5]

In this case (where the best alternative does not provide improvement) we change probability p only every k steps (we were most successful with $k = 40$ but this might be application dependent) to allow the algorithm time to converge.

The proposed approach combines ideas proposed in [37] for constructing a relation between the potential of improvement and the probability of an agent replacing its value assignment and for a dependent probability for replacing value assignments as in the Distributed Simulated Annealing algorithm [48, 36]. However, here the increasing probability of replacing an assignment is used for calculating the potential of improving the solution and not the time (as in [48]). It also includes monitored random selections that increase the exploration level further.

Our experimental study includes a justification for the use of the slope-dependent formula, which allows reasoning between the selection of multiple options of replacement probability, as opposed to a constant probability.

- The third approach stems from the DBA algorithm. We extend the possibilities for agents to break out of a quasi-local optima. However, we avoid the weaknesses of the DBA algorithm that made it inferior to DSA. While DBA is more explorative than other existing algorithms, its breakout mechanism changes the problem permanently. Thus, in many cases, after a few quasi-local optima are detected from which agents break out by increasing the weights of violated constraints they are involved in, the search space has changed so much that the states traversed by DBA are of low quality. We propose two alternative breakout methods. In the first, an agent detecting a quasi-local optima selects a random value

[4] In our experiments the most successful version had $p_A = 0.6$ and $p_B = 0.15$. Since SDP was the most successful heuristic found we present experiments that justify this selection in Section 8.

[5] In our experiments we used $p_C = 0.4$ and $p_D = 0.8$.

assignment for its variable. We call this method *Distributed Random Breakout* (*DRB*). This method attempts to replicate the success of random-restart methods that was reported in the case of centralized algorithms for solving constraint problems [47]. In the second proposed method we use the best value assignment found during the search from the agent's point of view. When the agent detects a quasi-local optimum it replaces its value assignment with this best value. Here we implement an approach of balancing exploration with exploitation of knowledge that was accumulated during search [9]. While the best assignment previously found may not be compatible with the current state of the other agents, it has evident potential to be part of a successful solution. We call this method *Distributed Breakout_Best Assignment Retrieval* (*DB_BAR*).

7.2.2. Algorithm-Independent Exploration Heuristics

We propose a *Random Restart* algorithm-independent heuristic to demonstrate the general ability of the ALS_DCOP framework to facilitate exploration in incomplete DCOP algorithms. In Random Restart exploration, the agents occasionally alter execution of their algorithm as if starting anew; this involves clearing any accumulated state and choosing a new, random assignment. Random Restart is well-suited for problems where the quality of local minima are heavily dependent on the starting assignment and where there are relatively many starting locations that can lead to low-cost solutions.

We consider two techniques to determine when to initiate exploration. The first triggers exploration after a fixed period of time, similar to the approach that we used in DSA-PPIRA and DSA-SDP. The second is to trigger exploration only when the search has converged to a local minimum, similar to the approach used with DRB. Detecting convergence can be achieved very naturally in ALS_DCOP because the root computes the total, global cost of solutions. We adopt a simple threshold-based convergence detector that decides that a local minimum has been reached when the computed global cost does not change for a number of steps T. When convergence has been detected, the root sends a Restart message to all of its children, who in turn propagate it to their children and so on. The Restart message instructs agents to randomly restart at the time when all agents receive the message; this is h steps after convergence is detected, where h is the height of the ALS tree. This guarantees that the exploration occurs simultaneously for all agents.

Random restart (and other algorithm-independent exploration heuristics) can be also incorporated into DCOP algorithms in an ad hoc fashion; for example, DSA_PPIRA and DRB both make use of random restarts. The goal here, however, is to consider how such heuristics can leverage the ALS_DCOP framework to add exploration in a way that is independent of the specific DCOP algorithm being used.

8. Experimental Evaluation

In order to demonstrate the impact of the ALS_DCOP framework on distributed local search, we present a a set of experiments that shows the effect of the proposed framework when combined with intensive exploration methods. We evaluate these methods on three different types of problems: unstructured random problems, structured graph-coloring problems, and realistic meeting-scheduling problems. Except

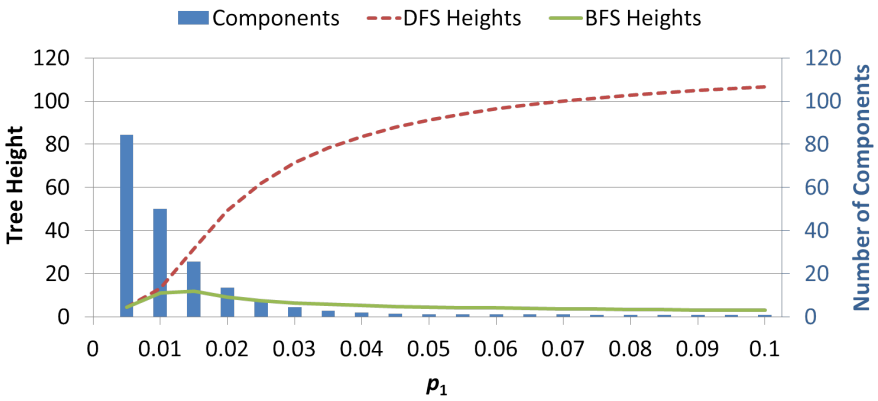

Figure 9: The average heights of the tallest BFS-trees and DFS-trees and the average number of connected components in unstructured random DCOPs as a function of the constraint density p_1.

where otherwise noted, each data point represents the average over 50 independently generated problems.

8.1. Unstructured Random Problems

The unstructured, uniformly random problems in our experiments were minimization random binary DCOPs in which each agent holds a single variable. The network of constraints in a problem was generated randomly by adding a constraint for each pair of agents/variables independently with probability p_1. The cost of any pair of assignments of values to a constrained pair of variables was selected uniformly at random from a finite, discrete range. Such uniformly random DCOPs with 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 (e.g., [14]). In our experiments we considered problems with $n = 120$ agents, $k = 10$ values in each domain, and costs chosen from the range $\{1, 2, \ldots, 10\}$.

8.1.1. BFS vs. DFS Spanning Trees

Our first set of experiments demonstrates the benefit of using a BFS-tree within the proposed framework. We generated 10000 uniformly random problem instances for each value of p_1 from 0.005 to 0.1 in increments of 0.005. For each problem instance we computed both breadth-first search and depth-first search spanning trees to find the connected components of the constraint network. Because the overhead of the ALS_DCOP framework is determined by the height of the tallest tree, for each problem instance we recorded the maximum tree height and averaged this maximum height across all 10000 problem instances with the same p_1 value. The results are shown in Figure 9. The average number of connected components for each density is also shown using columns plotted on the secondary y-axis.

DFS and BFS find trees of similar maximum heights when $p_1 = 0.005$ because the constraint graph is very sparse and most of the components have only a single agent;

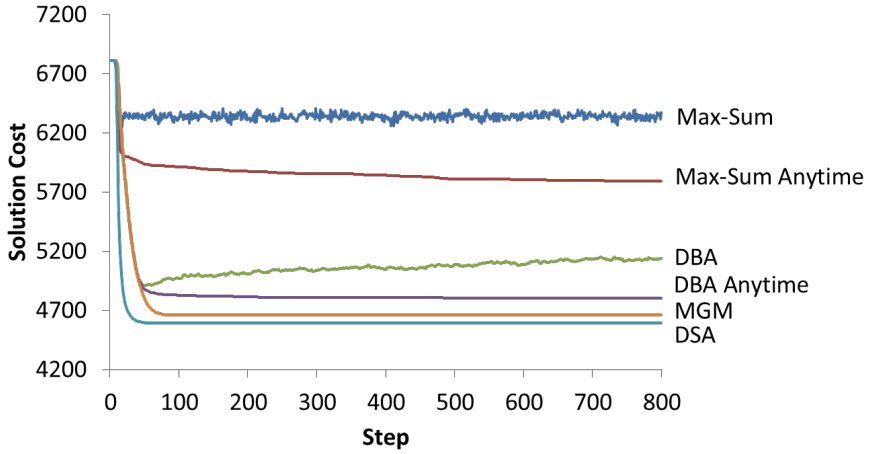

Figure 10: The cost in each step of standard incomplete algorithms solving random DCOPs, $p_1 = 0.1$.

on average the tallest trees for both approaches have height of 4.4. As p_1 increases the number of components drops rapidly. The average height of the tallest BFS-tree increases as the components become larger, peaking just under 12 for $p_1 = 0.015$. The average height of the tallest BFS-trees then decreases as p_1 increases further, averaging under 6 for $p_1 = 0.035$ (when there are fewer than 3 components on average), below 4 for $p_1 = 0.065$ (when there are only a 1.0398 components on average), and continuing to decline to an average maximum height of 3.0347 for $p_1 = 0.1$. This shows that when the constraint density is sufficiently high enough for there to be a single connected component, BFS tends to find trees that have only logarithmic height.

In contrast, the average height of the tallest DFS-trees continues to increase with the constraint density. Indeed, when $p_1 = 0.04$, the tallest DFS-tree averages a height of more than 83, and out of the 10000 randomly generated instances the tallest DFS-tree was never shorter than 63. The reason is that as the constraint graphs become denser, DFS is less likely to have to backtrack, resulting in taller trees. This is consistent with known results from random graph theory. For example Pósa [49] found that for a sufficiently large, finite constant c, the probability that random graphs with $p_1 = c\log(n)/n$ are Hamiltonian tends to 1 as n goes to infinity, and hence DFS may find a tree of height $n - 1$. It is true that other, shorter DFS-trees may also exist and that ordering heuristics (which we did not consider here) may help in finding shorter DFS trees. However, a DFS-tree starting from a given root will never be shorter than a BFS-tree starting from the same root, and no ordering heuristic is needed for the BFS. The simplicity of BFS and the superiority of its generally logarithmic-height trees make it clearly preferable for use in the ALS_DCOP framework.

8.1.2. Sparse Unstructured Problems

Our next set of experiments compared the performance of local search algorithms on sparse, unstructured random DCOPs with constraint density $p_1 = 0.1$. Figure 10

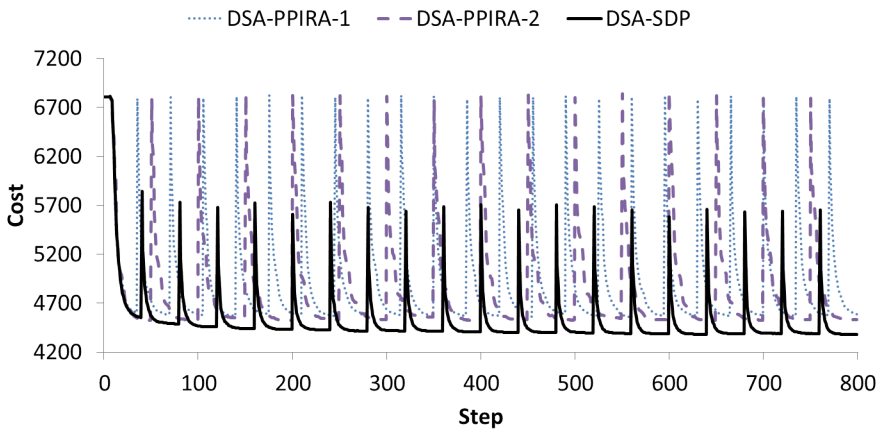

Figure 11: The cost in each step of DSA variants when solving random DCOPs, $p_1 = 0.1$.

presents the average cost in each step of the states found by the existing incomplete DCOP algorithms Max-Sum, DBA, MGM, and DSA. The curve of the average costs for each algorithm is labeled on the right by the name of the algorithm. We also present the average anytime costs of the best state found prior to each step for Max-Sum and DBA, as maintained by the ALS_DCOP mechanism; these are the curves labeled "Max-Sum Anytime" and "DBA Anytime," respectively.

Both DSA and MGM, being exploitive algorithms, produce smooth curves that decrease before converging. Their exploitive natures are verified by their anytime costs (not shown), which are very similar to the standard results depicted. MGM's standard cost curve and anytime cost curve match exactly, because in a monotonic algorithm the best solution found is also the most recent solution found. The two curves do not match exactly for DSA but the differences are not statistically significant. Neither algorithm performs significant exploration.

In contrast, both Max-Sum and DBA perform considerable exploration after an initial period of exploitation, as can be seen by their step-by-step costs first decreasing smoothly (exploitation), then increasing and decreasing (exploration). By comparing the current-state costs with the anytime costs, it is clear that ALS_DCOP allows them to find significantly better solutions, but these solutions are still of lower quality than those produced by MGM and DSA. One interesting observation is that while exploring, the average anytime cost at a step is better than the best average cost of the preceding steps. This is because different runs of the algorithms find higher-quality solutions in different steps. These lower- and higher-quality solutions offset each other in the average cost, but the average anytime cost aggregates the best solutions found in all previous steps preventing this offsetting; each time a run of an algorithm finds a better solution, the average anytime cost improves.

Figure 11 presents the results of DSA combined with the exploration methods proposed in this paper, PPIRA and SDP. We used two versions of the PPIRA method in our experiments that were found to be successful in parameter sensitivity checks. In

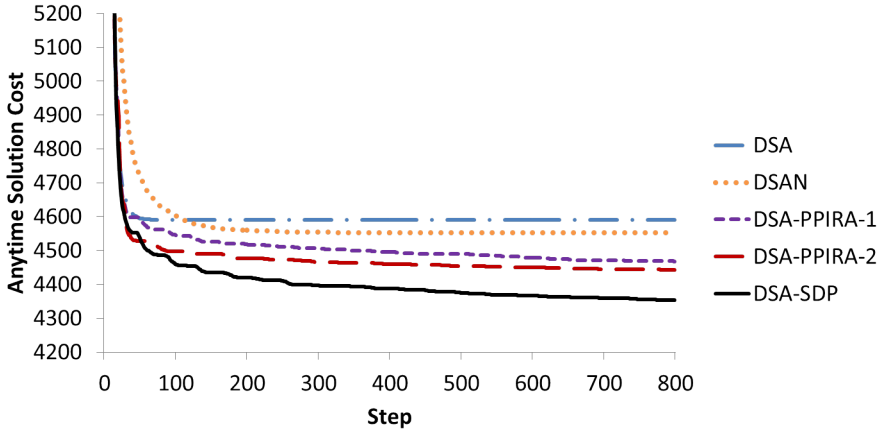

Figure 12: Anytime cost in each step of DSA variants when solving unstructured random DCOPs, $p_1 = 0.1$.

DSA-PPIRA-1, every 15 steps ($k = 15$) the probability of replacing an assignment was increased from $p = 0.4$ to $p^* = 0.8$ for $k^* = 5$ steps. Random assignment selections were performed by the agents every $r = 35$ steps. In DSA-PPIRA-2, the parameters were $k = 8$, $k^* = 5$, $p = 0.4$, $p^* = 0.9$, and $r = 50$. PPIRA-1 is more frequent in its random selections while PPIRA-2 is more frequent in its probability increases. DSA-SDP is described in detail in Section 7. The parameters of the algorithm in all our experiments were set to $p_A = 0.6$, $p_B = 0.15$, $p_C = 0.4$, $p_D = 0.8$. Later in this section we present experiments that justify the use of the slope-dependent formulas.

It is quite clear that the trends in the graphs in Figure 10 and Figure 11 are very different. The successful existing algorithms are exploitive. On the other hand, it is apparent that the versions of the DSA algorithm we proposed (and presented in Figure 11) perform intensive exploration.

Figure 12 presents the anytime results for DSA, DSAN, DSA-PPIRA-1, DSA-PPIRA-2, and DSA-SDP on the random setup presented in Figure 11. Although DSA converges more quickly than DSAN, it finds solutions of lower quality. This is expected because DSAN initially explores (so it does not converge as quickly to a local minimum) but that exploration allows it to possibly find better local minima. The three exploration heuristics that we combine with DSA outperform both DSA and DSAN while retaining the fast convergence of DSA. Among the PPIRA methods, it is apparent that PPIRA-2, which is the version that performs more frequent probability increases, is the version that performs better. Both of them are outperformed by the SDP method.

Figure 13 presents the anytime results for penalty-based algorithms solving the same problems as in Figures 11 and 12. MGM, MGM-2, DisPeL, and DBA are the existing algorithms while DRB and DB_BAR are the versions of the algorithm we propose in Section 7. The two variants of the algorithm we propose outperform the existing DBA-family of algorithms. DisPeL ultimately finds solutions of lower cost than DB_BAR, and of statistically similar costs as DRB. However, it is interesting to note that these three algorithms have very different convergence profiles: DB_BAR

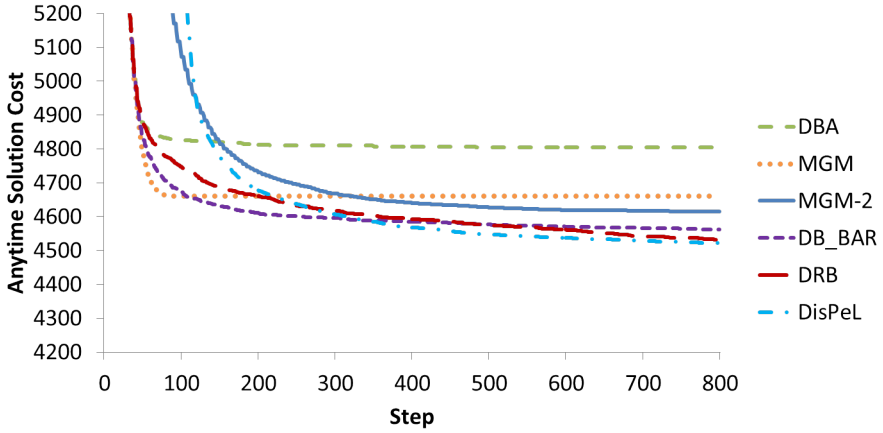

Figure 13: Anytime cost in each step of penalty-driven algorithms when solving unstructured random DCOPs, $p_1 = 0.1$.

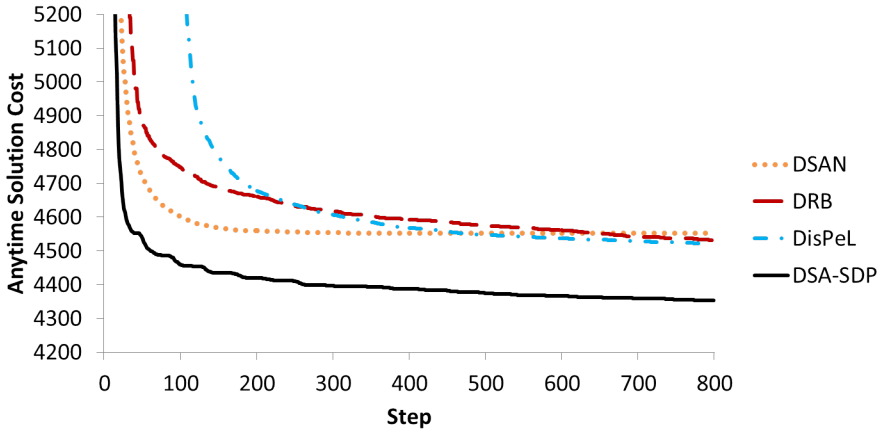

Figure 14: Anytime cost in each step of the best exploration methods compared with existing algorithms when solving unstructured random DCOPs, $p_1 = 0.1$.

finds lower-cost solutions very quickly while DRB finds them somewhat more slowly but ultimately finds better solutions. This is an example of the classic tradeoff between exploitive selection during the breakout procedure (DB_BAR) and the explorative, random selection of DRB. DisPeL takes the longest, partly due to the costly overhead of forming its tree, which has considerably greater height than the anytime tree.

Figure 14 presents the anytime results for the best performing existing algorithms (DisPeL and DSAN) and proposed algorithms (DRB and DSA-SDP) of the penalty-driven and DSA families of algorithms. DSAN converges much more quickly than DisPeL and DRB but finds a solution of ultimately lower quality. However, it is clear

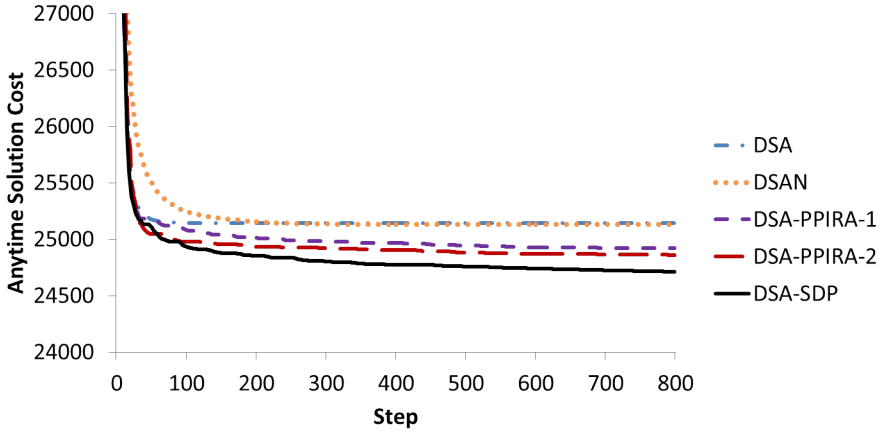

Figure 15: Anytime cost in each step of DSA variants when solving random DCOPs, $p_1 = 0.6$.

that the most successful algorithm is DSA-SDP.

8.1.3. Dense Unstructured Problems

Similar results are presented in Figures 15, 16, and 17 for the same algorithms solving dense problems ($p_1 = 0.6$). The relative performance of the DSA variants were similar to that in the sparse problems, with DSA-SDP finding the lowest cost solutions, but on denser problems DSAN did no better than DSA, despite taking longer to converge. Among the penalty-driven approaches, DRB and DB_BAR produced similar results to each other and outperformed the other algorithms. On these denser problems convergence was especially problematic for DisPeL, which took hundreds of steps to build its tree, and MGM-2, which was not even able to find a solution of equal quality to that found by MGM before termination. It is interesting that in this case the DSA approaches were superior, as seen in Figure 17: the best penalty-driven approach, DRB, only matched the solution quality found by DSAN (and hence DSA), and DSA-SDP found solutions of significantly lower cost than all other algorithms.

8.2. Graph Coloring Problems

In the next set of experiments the algorithms solved graph coloring problems, also with 120 agents. The number of colors in the problem (i.e., the domain size) was 3 and the density parameter $p_1 = 0.05$. As in standard graph coloring problems, we set the cost of each broken constraint (two adjacent variables with the same color) to one. These problems are known to be hard Max-CSP problems, i.e., beyond the phase transition between solvable and non-solvable problems [50].

For this set of experiments we maintain the three graph presentation of DSA variants, penalty-driven variants, and the best of our proposed approaches in comparison with the best existing algorithms. Figures 18, 19, and 20, respectively, depict these three graphs. For the DSA versions the performance of our proposed algorithms preserved the same trend observed for random DCOPs, DSAN found solutions that were

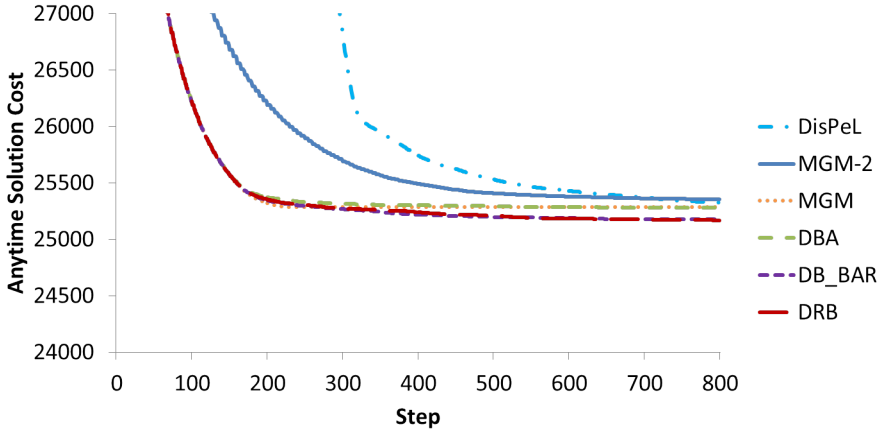

Figure 16: Anytime cost in each step of penalty-driven algorithms when solving random DCOPs, $p_1 = 0.6$.

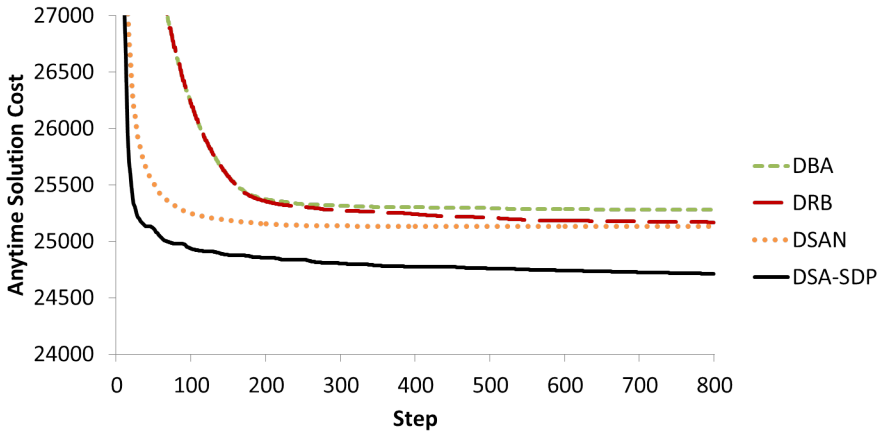

Figure 17: Anytime cost in each step of the best exploration methods compared with existing algorithms when solving random DCOPs, $p_1 = 0.6$.

not statistically significantly worse than DSA-SDP. This strong performance by DSAN is due to the anytime property, as we show in Section 8.5.1. The DSA-SDP version still dominated the other DSA variants, with a more apparent advantage over the other versions than with the dense, unstructured problems. We expand on this advantage of DSA-SDP later in this section.

In contrast to the results on random problems, the results of the penalty-driven variants when solving graph coloring problems reveal a clear advantage for DBA. Surprisingly, our two proposed variants perform the worst, finding worse solutions than even MGM. This suggests that there is considerable room for exploitation in graph coloring. However, there is also a need for exploration to find low cost solutions, as evidenced by

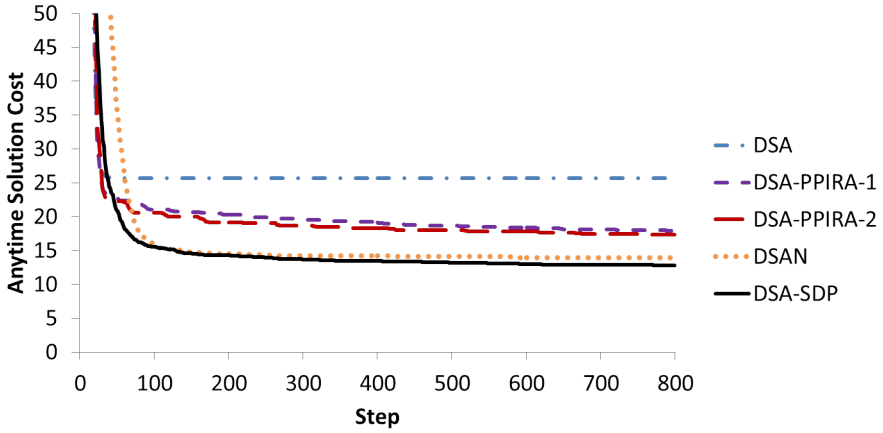

Figure 18: Anytime cost in each step of DSA variants when solving graph coloring problems.

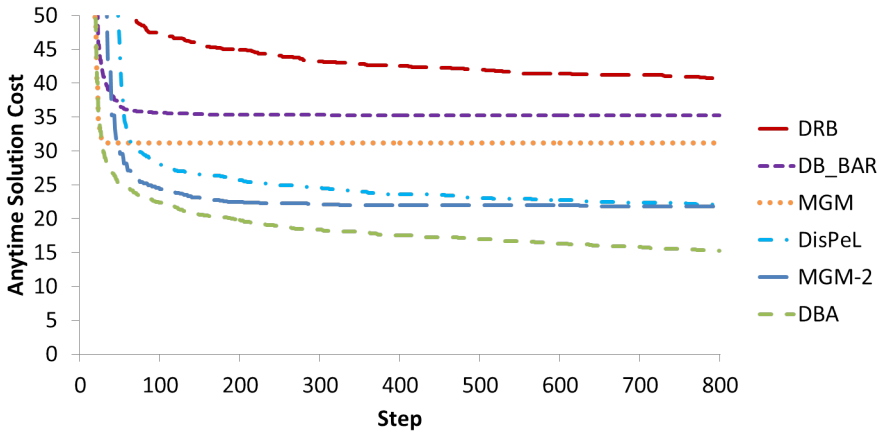

Figure 19: Anytime cost in each step of penalty-driven variants when solving graph coloring problems.

DisPeL and DBA outperforming the strictly exploitative MGM and MGM-2. The best algorithm found for graph coloring problems as for random problems was DSA-SDP (see Figure 20), converging faster than DBA and also finding solutions of better quality (although the trend suggests that DBA may be able to possibly find better solutions if given enough time).

8.3. Meeting Scheduling Problems

The next set of experiments was performed on realistic *Meeting Scheduling Problems* (MSPs) [51, 52, 53]. The meeting scheduling problem includes n agents that are trying to schedule m meetings. Each meeting m_i has $k_i \leq n$ specific agents that are intended to attend it. In addition, for every two meetings we randomly selected a *travel*

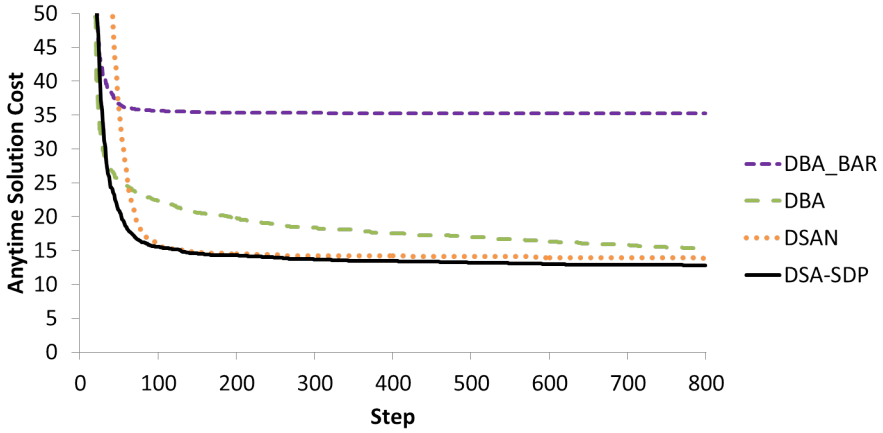

Figure 20: Anytime cost in each step of the best exploration methods compared with existing algorithms when solving graph coloring problems.

time that is required to get from the location of one meeting to the other. When the difference between the time-slots of two meetings with overlapping participants was less than the travel time, a scheduling conflict occurred for the overbooked agents who are scheduled to participate in both meetings. Constraints define the cost of scheduling conflicts and the agent's preferences for meetings being scheduled at each time. We designed the problem as a minimization problem. Thus, a scheduling conflict incurs a cost equal to the number of overbooked agents in the two meetings, and preferences for meeting times are represented by giving higher costs to meetings at less preferred times. The setup in this experiment included 90 agents and 20 meetings. There were 20 available time-slots for each meeting. The travel times between meetings were selected randomly between 6 and 10.

The results in Figure 21 show that our proposed heuristics outperform DSA and DSAN, and the differences between our algorithms are negligible. Figure 22 shows that in contrast to the graph coloring problems, DRB does very well in solving meeting scheduling problems. It is interesting to note that, like with graph coloring, exploration and exploitation both work well on meeting scheduling, as shown by the second-best penalty-driven approaches being DisPeL and MGM-2. However, DBA is not able to strike the right balance for these problem types and finds much worse solutions. Figure 23 shows that DSA-SDP ultimately finds very similar solutions to DRB, but converges to low-cost solutions much more quickly.

8.4. Algorithm-Independent Heuristic

Our next set of experiments tested the Random Restart algorithm-independent heuristic. To measure its effectiveness, we considered the improvement in the cost of the solution ultimately found relative to the cost of the standard solution found without the ALS_DCOP framework. We compared this improvement to that achieved with only the anytime framework and without the algorithm-independent heuristic. We used a

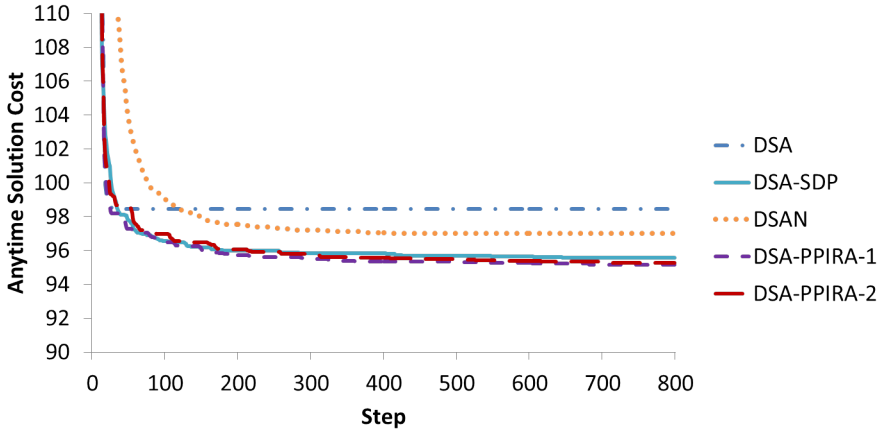

Figure 21: Anytime cost in each step of DSA variants when solving meeting scheduling problems.

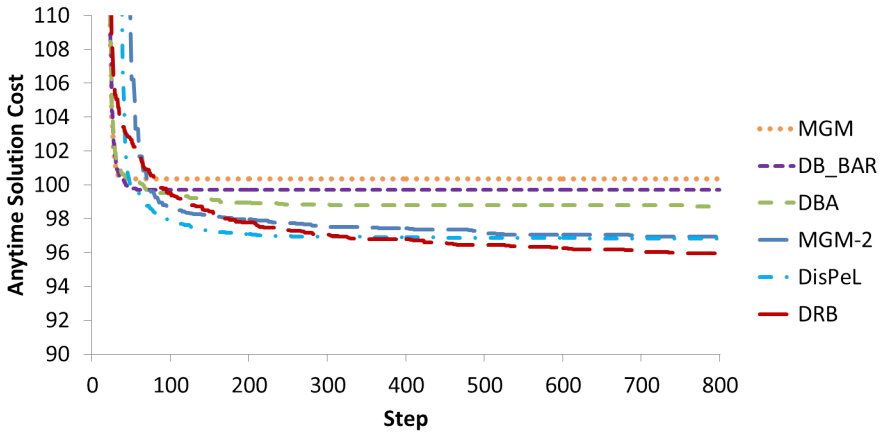

Figure 22: Anytime cost in each step of penalty-driven variants when solving meeting scheduling problems.

threshold of $T = 40$ steps to detect convergence for convergent exploration, and a period of 80 steps for periodic exploration.

Figure 24 shows the improvement in cost for the random restart heuristic on sparse, unstructured random problems ($p_1 = 0.1$). Convergent random restart can be seen to provide the same or better improvement than the anytime framework without additional exploration. It yields the same improvement in cost as the anytime framework for our proposed DSA variants, all variants of DBA, and Max-Sum. This is because these algorithms perform a considerable amount of exploration already, and hence convergence is not detected and the random restart exploration is never triggered. In contrast, the algorithms that are monotonic or nearly-monotonic on these problems (DSA, DSAN, MGM, and MGM-2) show no improvement in cost from the

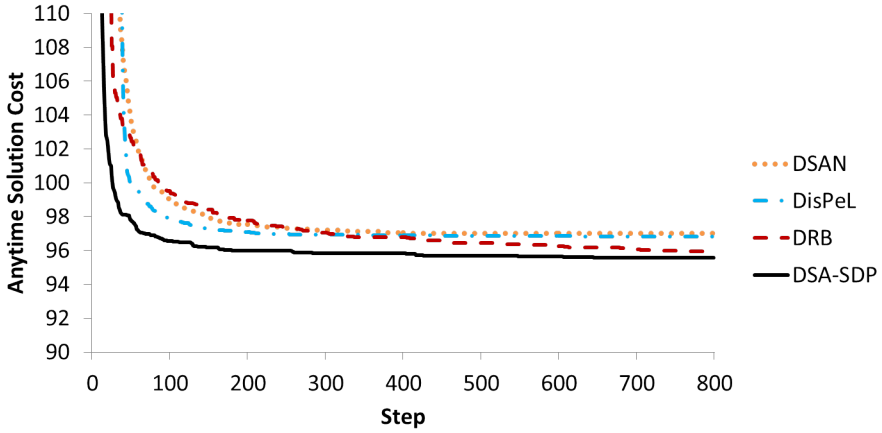

Figure 23: Anytime cost in each step of the best exploration methods compared with existing algorithms when solving meeting scheduling problems.

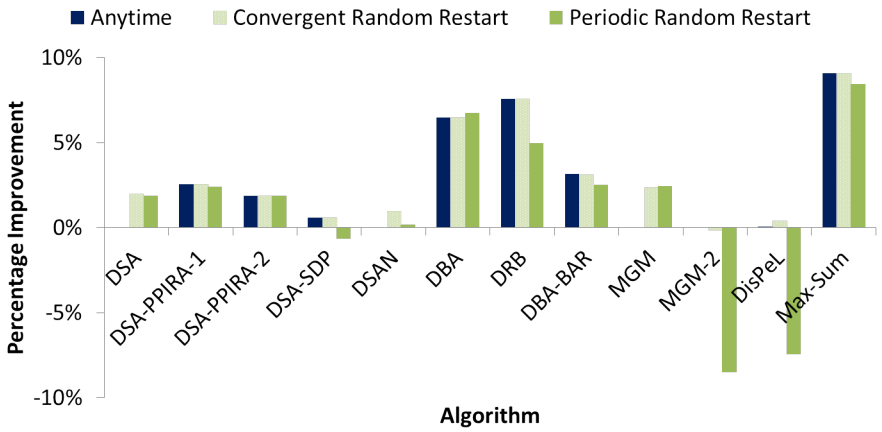

Figure 24: Percentage improvement of the solution cost due to the anytime mechanism alone and random restarts with the anytime mechanism on sparse unstructured random problems ($p_1 = 0.1$).

ALS_DCOP framework because they are already anytime. When the algorithm converges quickly (DSA, DSAN, and MGM), convergent random restart improves performance by allowing it to sample multiple local minima over the course of execution and choose the one with lowest cost. When it converges slowly (MGM-2) convergent random restart has little or no effect because there are correspondingly fewer opportunities for it to trigger exploration.

In contrast to the "safe" amounts of exploration provided by convergent random restart, periodic random restart can have negative effects on the quality of the solution found, indicated by smaller improvement than with the anytime framework. In ex-

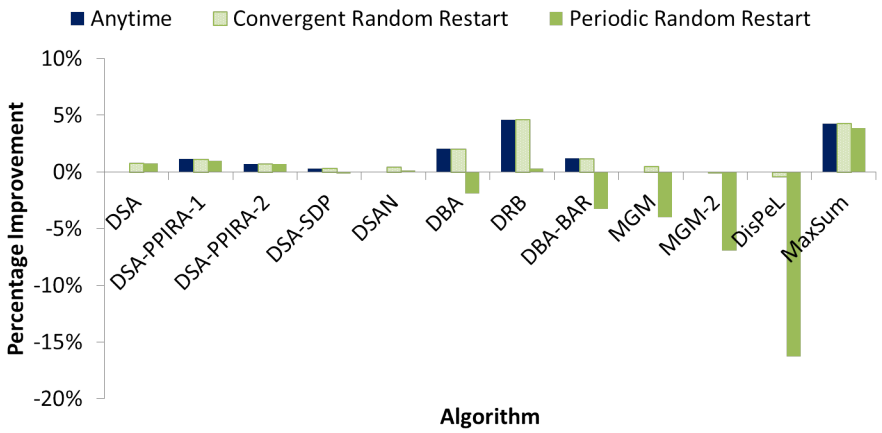

Figure 25: Percentage improvement of the solution cost due to the anytime mechanism alone and random restarts with the anytime mechanism on dense unstructured random problems ($p_1 = 0.6$).

treme examples (MGM-2 and DisPeL), periodic random restart even worsens performance relative to the original algorithm without ALS_DCOP at all. This is because it forces exploration without considering the local state of the algorithm. For the slowly-converging algorithms like MGM-2 and DisPeL, this means that they are interrupted before being able to converge, thus worsening performance. It is also harmful to algorithms like DSA-SDP or DRB that already perform algorithm-specific exploration. It is helpful for rapidly-converging monotonic algorithms like DSA and MGM, but not significantly more so than convergent random restart.

Figure 25 shows the improvement in costs on dense unstructured problems ($p_1 = 0.6$). The results are broadly similar to those for the sparse problems, except that periodic random restarts lead to worse solution quality for a wider range of algorithms, including DBA, DBA_BAR, and MGM in addition to MGM-2 and DisPeL. In addition, it worsens the performance of DRB and DisPeL more than it did in the sparse problems. This occurs because convergence in dense problems is slower than in sparse problems, as suggested from a comparison of Figures 13 and 16. By interrupting the local search algorithms before they can converge to a local minimum, the random restart heuristic increases solution cost.

Figure 26 shows the improvement in costs on graph coloring problems. The results are broadly similar to those for the unstructured problems, although MGM-2 and DisPeL both converge quickly enough that both convergent random restart and periodic random restart provide improvement. A notable difference from the results with unstructured problems is that periodic random restart greatly improves the solution found by DBA_BAR. This is likely because DBA_BAR gets trapped in cycles on graph coloring problems, with agents reaching quasi-local minima and reverting to their best previously value assignment. Because it is a cycle, costs do not converge and hence convergent random restart shows no effect, but periodic random restart is able to allow the algorithm to explore other regions of the search space, leading to improved

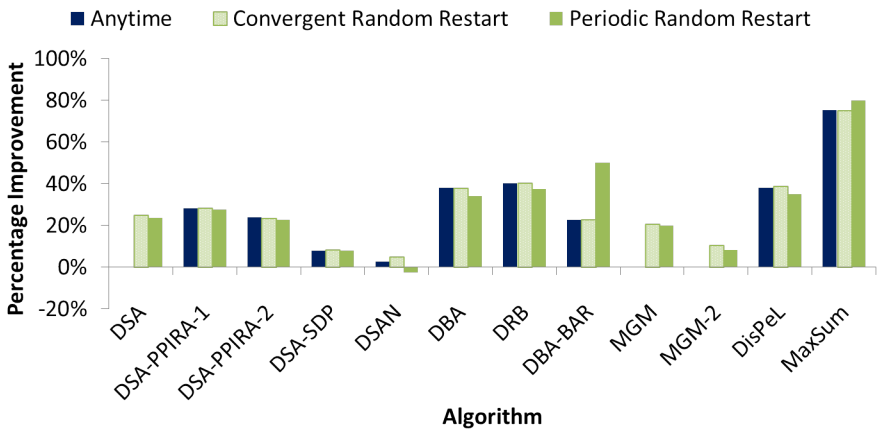

Figure 26: Percentage improvement of the solution cost due to the anytime mechanism alone and random restarts with the anytime mechanism on graph coloring problems ($p_1 = 0.05$).

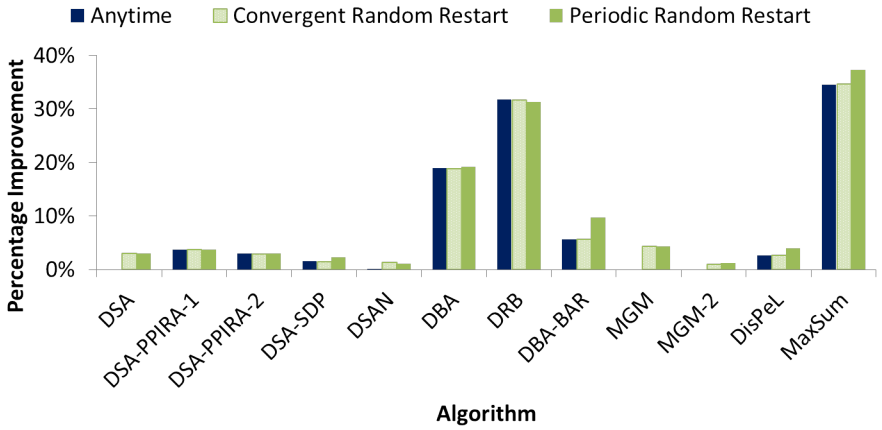

Figure 27: Percentage improvement of the solution cost due to the anytime mechanism alone and random restarts with the anytime mechanism on meeting scheduling problems.

performance.

The results on the meeting scheduling problems are shown in Figure 27. These are qualitatively similar to those for the graph coloring problems.

8.5. Analysis

8.5.1. Statistical Significance

In order to validate the statistical significance of the presented results we ran a paired difference test (t-test) comparing the standard results of the algorithms with their anytime results obtained via the framework proposed in this paper. The results

Problem Type			DSA			
	Max-Sum	DSAN	DSA	PPIRA-1	PPIRA-2	SDP
Sparse Unstructured	+	−	−	+	+	+
Dense Unstructured	+	−	−	+	+	+
Graph Coloring	+	+	−	+	+	+
Meeting Scheduling	+	−	−	+	+	+

Table 1: Paired difference statistical significance results for Max-Sum, DSAN, and DSA-family algorithms.

		DBA				
Problem Type	DisPeL	MGM	MGM-2	DBA	DRB	DBA_BAR
Sparse Unstructured	+	−	−	+	+	+
Dense Unstructured	−	−	−	+	+	+
Graph Coloring	+	−	−	+	+	+
Meeting Scheduling	+	−	−	+	+	+

Table 2: Paired difference statistical significance results for penalty-driven algorithms.

are presented in Tables 1 and 2. For each problem type and algorithm there is a '+' sign if the difference between the anytime results and the standard results were found to be significant (p-value ≤ 0.01) and a '−' sign if not. The results of the statistical significance checks were conclusive. The anytime solution cost was found to be significantly better than the standard result for every explorative algorithm on every type of problem. On the other hand, the differences for the monotonic algorithms (MGM, MGM-2 and DSA) were not found to be significant.

DSAN and DisPeL occupied a middle ground. For DSAN, the average anytime costs were only significantly better for the graph coloring problems, but as seen in Figure 18, this is exactly the case where DSAN found very good solutions comparable to DSA-SDP. It seems that in graph coloring, the exploration performed by DSAN is helpful for finding good states that have lower cost than those to which DSAN eventually converges. On other problem types, DSAN did not use the anytime framework to improve solution quality. In contrast, DisPeL utilized the anytime mechanism to achieve significantly better results through exploration on the sparse unstructured, graph coloring, and meeting scheduling problems, while on dense unstructured problems its exploration was able to generally guide its search toward improving solutions to the extent that the anytime framework did not significantly improve the solution quality. These results emphasize the strong relation between the anytime property and explorative search methods.

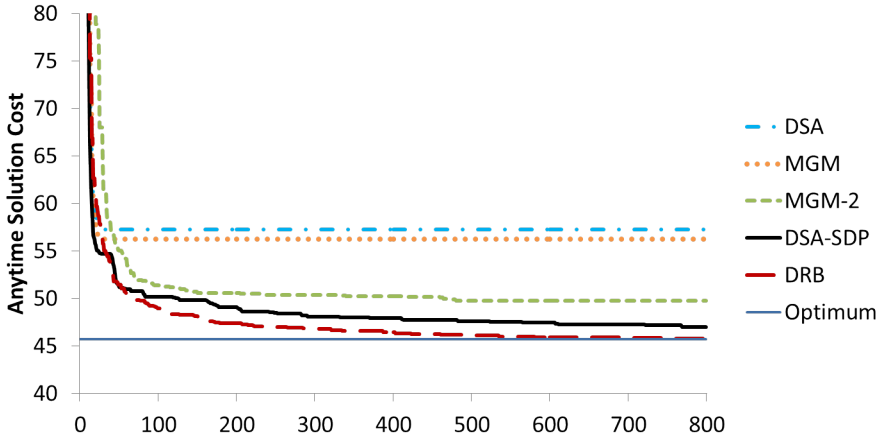

Figure 28: Anytime cost in each step of selective local algorithms in comparison with the optimal solution when solving small random DCOPs, $p_1 = 0.3$.

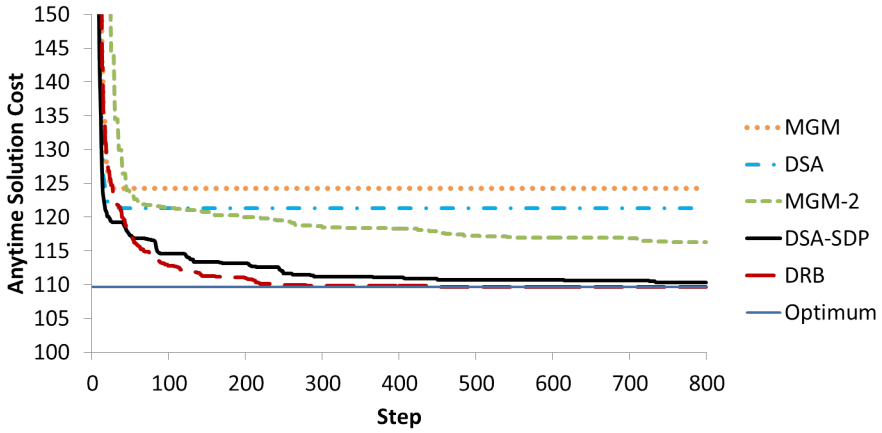

Figure 29: Anytime cost in each step of selective local algorithms in comparison with the optimal solution when solving small random DCOPs, $p_1 = 0.7$.

8.5.2. Comparison with Optimal

In order to evaluate the relation between the quality of the solutions produced by our proposed methods and the optimal solution, we performed an experiment on a much smaller problem scenario in which we were able to find the optimal solutions using an exhaustive search algorithm. Figures 28 and 29 include experiments on random problems with 10 agents. To avoid multiple components, we chose higher constraint densities than in the experiments performed on larger random problems. The explorative heuristics proposed in this paper outperformed the existing monotonic algorithms and

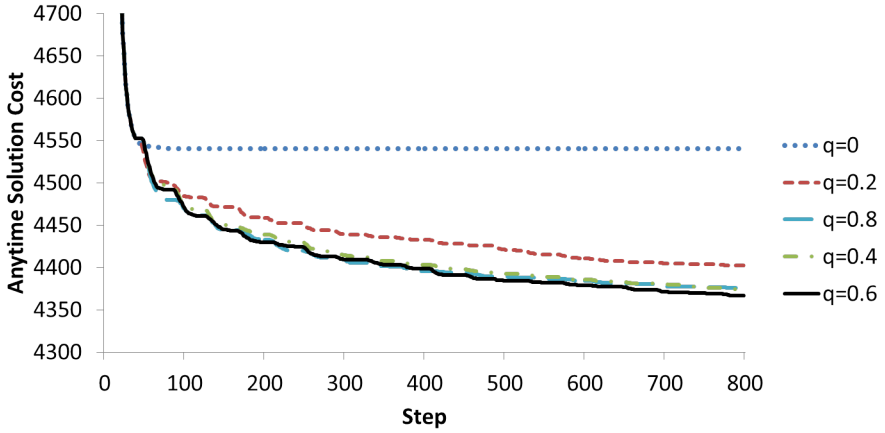

Figure 30: Anytime cost in each step of the DSA-SDP method with a constant probability for periodically selecting a non-improving value assignment.

in fact found solutions very close to global optima most of the time. This was particularly true of DRB which converged faster and ultimately found better solutions even than DSA-SDP.

It is interesting to note that DRB was able to converge to near-optimal solutions faster on the denser problems than it was on the sparser problems. In these cases the quasi-local optima are likely to involve most if not all of the agents. However, it is clear that the quality of the quasi-local optima that are reached depends heavily on the initial conditions, because the exploration of DRB upon reaching quasi-local optima leads to far better solutions than the purely monotonic MGM. DRB also outperformed MGM-2, which converges to provably better solutions than MGM, and due to MGM-2's extremely slow rate of improvement, it is not even clear if it would have eventually converged to the globally optimal solution.

8.5.3. Examination of DSA-SDP

Our experiments demonstrate that DSA-SDP consistently does as well as or better than the other existing and proposed algorithms across all large problem scenarios. Thus, we present a set of experiments that investigate this success. The DSA-SDP algorithm implements two key innovations. First, the probability of an agent updating its value is dependent on the magnitude of the local-cost change. Second, when an agent cannot improve its local cost, it periodically chooses the next-best value stochastically, so that in most cases only a subset of the agents make such changes at once. This is in contrast to DSA-PPIRA, in which all agents periodically choose a random value (and hence may increase their local costs), or DSAN, in which agents always (not periodically) make a stochastic choice to increase their local cost if that is their best alternative. The following set of experiments investigated the importance of the combination of these two ideas in DSA-SDP.

Figure 30 presents the results of DSA-SDP solving random problems with density $p_1 = 0.1$ with a small change. The decision whether to change to a non-improving

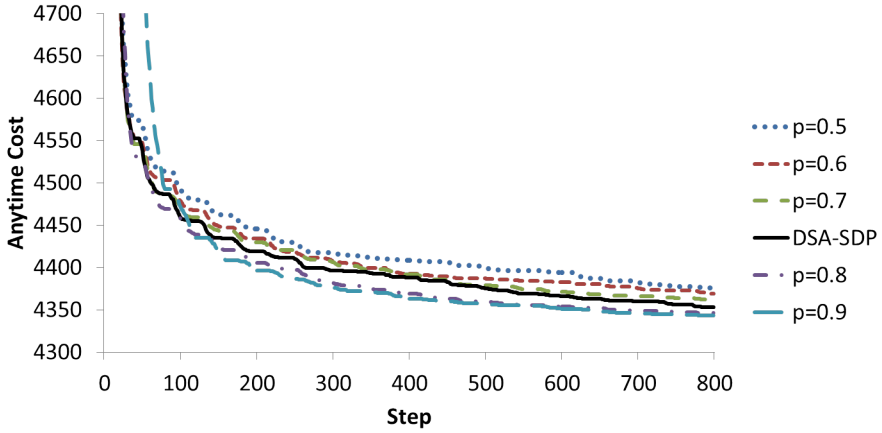

Figure 31: Anytime cost in each step of DSA with different probabilities for replacing a value assignment and a constant probability for periodically selecting a non-improving value assignment.

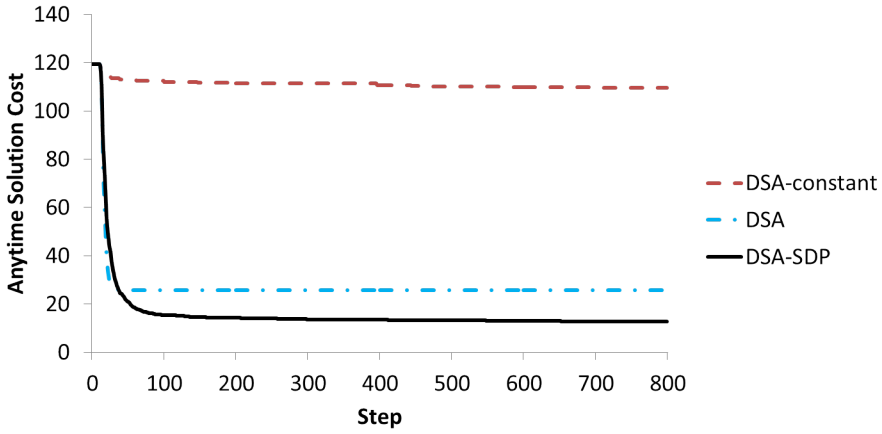

Figure 32: Anytime cost in each step of DSA, DSA with a constant probability for periodically selecting a non-improving value assignment, and DSA-SDP solving graph coloring problems.

value assignment every 40 steps of the algorithm was performed with a constant probability q. This experiment reveals that the algorithm is most successful when $q = 0.6$. Figure 31 presents a comparison of DSA-SDP with versions of the algorithm in which $q = 0.6$ and the parameter p, for deciding whether to replace a value assignment with a value that is at least as good, was fixed as in standard DSA. The results demonstrate that on random uniform problems, fixed probabilities for both decisions are enough to produce similar results to the results we obtain when using DSA-SDP.

On the other hand, Figures 32 and 33 present the results of DSA and DSA-SDP in comparison with the DSA version with constant probability for periodically selecting a non-improving value assignment, which was found most successful in the experiments presented in Figure 31 (termed DSA-constant in these figures). The results demonstrate

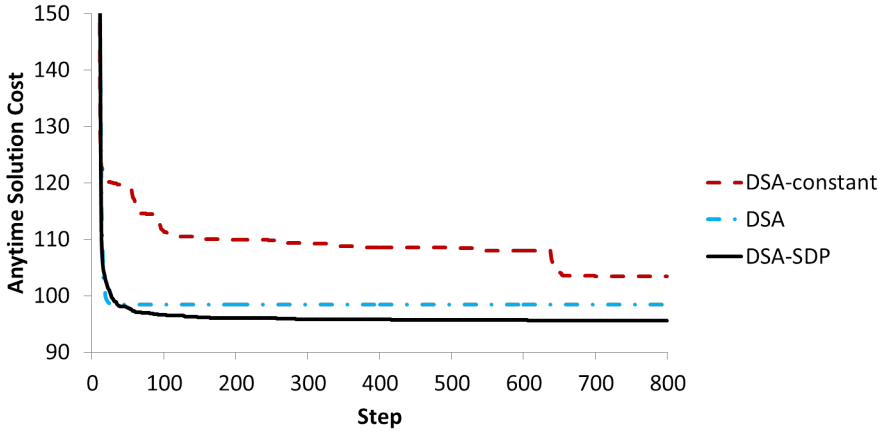

Figure 33: Anytime cost in each step of DSA, DSA with a constant probability for periodically selecting a non-improving value assignment, and DSA-SDP solving meeting scheduling problems.

that the DSA-constant version (that was found most successful for uniform problems) performs poorly on structured and realistic problems. On the other hand, the DSA-SDP algorithm performs well on all the problem scenarios used in our experiments. Thus, the slope-dependent approach for stochastic decisions, which we introduced in DSA-SDP, is apparently sensitive to the problem structure.

9. Discussion

While most of the graphs we present in Section 8 compare the different exploration heuristics we propose in Section 7, the most important contribution in this study is the significant improvement when combining explorative algorithms with the proposed anytime framework. All graphs presenting the anytime results show monotonic improvement in contrast to the standard results (as depicted in Figure 11, for example). This improvement over the standard results can be seen in Figures 24 – 27, and the significance of this improvement was validated in the results presented in Tables 1 and 2.

The explorative heuristics we proposed in this study share a common property. They all behave in an exploitive manner with bursts of explorative actions. In the methods combined with DSA, these explorative bursts are periodic. In the methods combined with DBA, the bursts are triggered by a local identification of idleness. The algorithm-independent exploration heuristics allow either type of triggering. The results show dominance of methods combined with DSA over methods combined with DBA, and dominance of algorithm-specific exploration over algorithm-independent exploration. These advantages are more apparent when solving random, unstructured problems. It seems that the local idleness detection (of quasi-local minima) performed by DBA algorithms is more effective in problems with structure.

Among the DSA versions, the DSA-SDP algorithm dominated in most problem scenarios and was consistently a strong performer. Our investigation of this algorithm

revealed that for specific problems it is possible to use constant probabilities for making the same decisions made by agents in DSA-SDP, i.e., the decisions on whether to replace a value assignment for the best alternative or to periodically change to a non-improving value assignment. However, fixed probabilities tuned for a specific problem type can result in arbitrarily bad performance when solving problems of different types. On the other hand, the decisions made by DSA-SDP that are dependent on the potential for improvement by replacing value assignments are robust over all the problem scenarios we have experimented with. It is important to note that like the DSA-PPIRA versions, DSA-SDP includes a number of parameters that were set following a sensitivity analysis; however, in contrast to other exploration methods, DSA-SDP is much less sensitive to small variations in these parameters because these parameters define ranges over which DSA-SDP bases its behavior on the specific local improvements that are possible during execution. This makes it much easier to choose parameters for DSA-SDP that are broadly acceptable over a wide range of problem types.

10. Conclusions

Distributed Constraint Optimization Problems (DCOPs) can be used to model many realistic combinatorial problems that are distributed by nature. The growing interest in this field has motivated intensive study in recent years on search algorithms for solving DCOPs. Since DCOPs are NP-hard optimization problems, complete algorithms are useful only for small problems. Larger problems like the ones studied in our experiments require incomplete methods.

Distributed local search algorithms were originally proposed for Distributed Constraint Satisfaction Problems and subsequently applied for DCOPs [7]. However, these algorithms failed to report the best state traversed by the algorithm, due to the challenge in evaluating global cost from the private, local costs of individual agents.

To meet this challenge, we proposed ALS_DCOP, a general framework for performing distributed local search in DCOPs that provides them with the *anytime* property. In the proposed framework, agents use a spanning tree structure in order to accumulate the costs of a state of the system to the root agent, which compares the cost of each state with the cost of the best state found so far and propagates the step index of a new best state, once it is found, to all other agents. At the end of the run the agents hold the best state that was traversed by the algorithm.

Apart from a small number of idle steps at the end of the run of the algorithm (twice the height of the spanning tree), the framework does not require any additional slowdown in the performance of the algorithm. In contrast to complete algorithms that use a pseudo-tree, the tree used in ALS_DCOP can be a *Breadth First Search* (BFS) tree. Thus, the height of the tree is expected to be small. In terms of network load, the only messages used in the ALS_DCOP framework are the algorithm's messages (i.e., no additional messages are required by the framework). Agents are required to use small (linear in the worst case) additional space.

Most existing local search algorithms for DCOPs are either monotonic (completely exploitive) or perform limited exploration. Thus, there is a limited benefit when combining the ALS_DCOP framework with these algorithms. However, we demonstrate

that the framework enhances existing explorative algorithms such as DBA and Max-Sum. In order to demonstrate the full potential of the framework we proposed extreme explorative methods, which are combined with existing local search algorithms (DSA and DBA), and two algorithm-independent exploration methods that can be combined with any existing incomplete DCOP algorithms. Our results demonstrate the advantage of the combination of exploration and the anytime property over standard local search.

[1] R. Zivan, Anytime local search for distributed constraint optimization, in: Proc. AAAI 2008, Chicago, IL, USA, 2008, pp. 393–398.

[2] S. M. Ali, S. Koenig, M. Tambe, Preprocessing techniques for accelerating the DCOP algorithm ADOPT, in: Proc. of AAMAS 2005, Utrecht, The Netherlands, 2005, pp. 1041–1048.

[3] R. Mailer, V. Lesser, Solving distributed constraint optimization problems using cooperative mediation, in: Proc. AAMAS 2004, New York, USA, 2004, pp. 438–445.

[4] P. J. Modi, W. Shen, M. Tambe, M. Yokoo, ADOPT: Asynchronous distributed constraint optimization with quality guarantees, Artificial Intelligence 161:1-2 (2005) 149–180.

[5] A. Petcu, B. Faltings, A scalable method for multi-agent constraint optimization, in: Proc. IJCAI 2005, 2005, pp. 266–271.

[6] M. C. Silaghi, M. Yokoo, Nogood based asynchronous distributed optimization (ADOPT-ng), in: Proc. AAMAS 2006, Hakodate, Japan, 2006, pp. 1389–1396.

[7] W. Zhang, Z. Xing, G. Wang, L. Wittenburg, Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks, Artificial Intelligence 161(1-2) (2005) 55–88.

[8] A. Farinelli, A. Rogers, A. Petcu, N. R. Jennings, Decentralised coordination of low-power embedded devices using the max-sum algorithm, in: Proc. AAMAS 2008, Estoril, Portugal, 2008, pp. 639–646.

[9] M. E. Taylor, M. Jain, P. Tandon, M. Tambe, Using DCOPs to balance exploration and exploitation in time-critical domains, in: Proc. Workshop on Distributed Constraint Reasoning DCR-09 IJCAI, Pasedena, CA, 2009.

[10] A. Petcu, B. Faltings, A value ordering heuristic for distributed resource allocation, in: Proc. CSCLP 2004, Lausanne, Switzerland, 2004.

[11] G. Solotorevsky, E. Gudes, A. Meisels, Modeling and solving distributed constraint satisfaction problems (DCSPs), in: Proc. CP 2006 (short paper), Cambridge, MA, USA, 1996, pp. 561–2.

[12] E. Kaplansky, A. Meisels, Distributed personnel scheduling - negotiation among scheduling agents, Annals OR 155 (1) (2007) 227–255.

[13] Y. Chong, Y. Hamadi, Distributed log-based reconciliation, in: Proc. ECAI 2006, 2006, pp. 108–113.

[14] A. Gershman, A. Meisels, R. Zivan, Asynchronous forward bounding, J. of Artificial Intelligence Research (JAIR) 34 (2009) 25–46.

[15] R. T. Maheswaran, J. P. Pearce, M. Tambe, Distributed algorithms for DCOP: A graphical-game-based approach, in: Proc. PDCS 2004, 2004, pp. 432–439.

[16] W. T. L. Teacy, A. Farinelli, N. J. Grabham, P. Padhy, A. Rogers, N. R. Jennings, Max-sum decentralized coordination for sensor systems, in: Proc. AAMAS 2008, 2008, pp. 1697–1698.

[17] W. Yeoh, X. Sun, S. Koenig, Trading off solution quality for faster computation in dcop search algorithms, in: Proc. IJCAI 2009, 2009, pp. 354–360.

[18] K. Hirayama, M. Yokoo, The distributed breakout algorithms, Artificial Intelligence 161:1-2 (2005) 89–116.

[19] R. Zivan, R. Glinton, K. Sycara, Distributed constraint optimization for large teams of mobile sensing agents, in: Proc. IAT 2009, Milan Italy, 2009, pp. 347–354.

[20] M. Jain, M. E. Taylor, M. Yokoo, M. Tambe, DCOPs meet the real world: Exploring unknown reward matrices with applications to mobile sensor networks, in: Proc. IJCAI 2009, Pasadena, CA, USA, 2009, pp. 181–186.

[21] C. Kiekintveld, Z. Yin, A. Kumar, M. Tambe, Asynchronous algorithms for approximate distributed constraint optimization with quality bounds, in: Proc. AAMAS 2010, Toronto, Canada, 2010, pp. 133–140.

[22] F. Glover, M. Laguna, Tabu search, Kluwer Academic Publishers, 1997.

[23] A. Schaerf, Local search techniques for large high-school timetabling problems, IEEE Transactions on Systems, Man, and Cybernetics— Part A: Systems and Humans 29 (4) (1999) 368–377.

[24] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.

[25] S. Zilberstein, Using anytime algorithms in intelligent systems, AI Magazine 17 (3) (1996) 73–83.

[26] S. Dolev, Self Stabilization, MIT Press, 2000.

[27] K. Hirayama, M. Yokoo, Distributed partial constraint satisfaction problem, in: Proc. ICMAS 1996, 1996.

[28] W. Yeoh, A. Felner, S. Koenig, BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm, J. of Artificial Intelligence Research (JAIR) 38 (2010) 85–133.

[29] S. M. Aji, R. J. McEliece, The generalized distributive law, Information Theory, IEEE Transactions 46(2) (2000) 325–343.

[30] I. Brito, P. Meseguer, Improving DPOP with function filtering, in: Proc. AAMAS 2010, Toronto, Canada, 2010, pp. 141–148.

[31] M. Vinyals, J. A. Rodríguez-Aguilar, J. Cerquides, Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law, Autonomous Agents and Multi-Agent Systems 22 (3) (2011) 439–464.

[32] T. Grinshpoun, A. Meisels, Completeness and performance of the APO algorithm, J. of Artificial Intelligence Research (JAIR) 33 (2008) 223–258.

[33] M. Basharu, I. Arana, H. Ahriz, Solving DisCSPs with penalty driven search, in: Proceedings of the 20th national conference on Artificial intelligence - Volume 1, AAAI'05, AAAI Press, 2005, pp. 47–52.

[34] J. P. Pearce, M. Tambe, Quality guarantees on k-optimal solutions for distributed constraint optimization problems., in: Proc. IJCAI 2007, Hyderabad, India, 2007.

[35] M. Vinyals, E. Shieh, J. Cerquides, J. A. Rodriguez-Aguilar, Z. Yin, M. Tambe, E. Bowring, Quality guarantees for region optimal DCOP algorithms, in: Proc. AAMAS 2011, Taipei, 2011, pp. 133–140.

[36] M. Smith, R. Mailler, Getting what you pay for: Is exploration in distributed hill climbing really worth it?, in: IAT, 2010, pp. 319–326.

[37] M. Smith, S. Sen, R. Mailler, Adaptive and non-adaptive distribution functions for DSA, in: PRIMA, 2010, pp. 58–73.

[38] A. Grubshtein, R. Zivan, T. Grinshpoun, A. Meisels, Local search for distributed asymmetric optimization, in: Proc. AAMAS 2010, 2010, pp. 1015–1022.

[39] M. Yokoo, Algorithms for distributed constraint satisfaction problems: A review, Autonomous Agents and Multi-Agent Systems 3 (2000) 198–212.

[40] M. Yokoo, Distributed Constraint Satisfaction, Foundations of Cooperation in Multi-agent Systems, Springer Verlag, 2000.

[41] F. R. Kschischang, B. J. Frey, H. A. Loeliger, Factor graphs and the sum-product algorithm, IEEE Transactions on Information Theory 47(2) (2001) 498–519.

[42] A. Rogers, A. Farinelli, R. Stranders, N. R. Jennings, Bounded approximate decentralised coordination via the max-sum algorithm, Artif. Intell. 175 (2) (2011) 730–759.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.

[44] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Series, 1997.

[45] R. Greenstadt, B. J. Grosz, M. D. Smith, SSDPOP: Improving the privacy of DCOP with secret sharing, in: Distributed Constraint Reasoning Workshop (DCR), CP 2007, Providence, RI, USA, 2007.

[46] R. Zivan, H. Peled, Max/min-sum distributed constraint optimization through value propagation on an alternating DAG, in: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '12, 2012, pp. 265–272.

[47] H. Wu, P. van Beek, On universal restart strategies for backtracking search, in: Proc. CP 2007, 2007, pp. 681–695.

[48] M. Arshad, M. C. Silaghi, Distributed simulated annealing, Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, Frontiers in Artificial Intelligence and Applications series 112.

[49] L. Pósa, Hamiltonian circuits in random graphs, Discrete Mathematics 14 (4) (1976) 359 – 364.

[50] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, T. Walsh, Random constraint satisfaction: Flaws and structure, Constraints 6 (4) (2001) 345–372.

[51] I. Gent, T. Walsh, CSPLib: a benchmark library for constraints, Tech. rep., Technical report APES-09-1999, available from http://csplib.cs.strath.ac.uk/. A shorter version appears in Proc. CP 1999, Alexandria, VA, USA, (1999).

[52] A. Meisels, O. Lavee, Using additional information in DisCSP search, in: Proc. Workshop on Distributed Constraint Reasoning (DCR), AAMAS 2004, Toronto, Canada, 2004.

[53] J. Modi, M. Veloso, Multiagent meeting scheduling with rescheduling, in: Proc. Workshop on Distributed Constraint Reasoning (DCR), CP 2004, Toronto, 2004.