Distributed Breakout: Beyond Satisfaction

Steven Okamoto and Roie Zivan and Aviv Nahon
Industrial Engineering and Management Department
Ben-Gurion University of the Negev
Beer-Sheva, Israel
\{okamotos,zivanr,nahona\}@bgu.ac.il

Abstract

The Distributed Breakout Algorithm (DBA) is a local search algorithm that was originally designed to solve DisCSPs and DisMaxCSPs. Extending it to general-valued DCOPs requires three design choices: manner of modifying base costs (multiplicative weights or additive penalties); definition of constraint violation (non-zero cost, non-minimum cost, and maximum cost); and scope of modifying cost tables during breakout (entry, row, column, or table).

We propose Generalized DBA (GDBA) to span the 24 combinations in the three dimensions. In our theoretical analysis we prove that some variants of GDBA are equivalent for certain problems, and prove that other variants may find suboptimal solutions even on tree topologies where DBA is complete. Our extensive empirical evaluation on various benchmarks shows that in practice, GDBA is capable of finding solutions of equal or significantly lower cost than alternative heuristic approaches (including DSA).

1 Introduction

Distributed constraint satisfaction and optimization problems (DisCSPs and DCOPs) are common frameworks for representing multiagent coordination problems that are distributed by nature, such as target tracking in sensor networks [Zhang et al., 2005] and scheduling meetings in offices [Maheswaran et al., 2004b]. Because they are NP-hard, considerable research effort has been devoted to developing incomplete algorithms for finding good solutions quickly [Yokoo and Hirayama, 1996; Maheswaran et al., 2004a; Zhang et al., 2005; Basharu et al., 2005; Farinelli et al., 2008; Smith and Mailler, 2010]. Local search algorithms, e.g., DSA [Zhang et al., 2005] and MGM [Maheswaran et al., 2004a], are simple incomplete algorithms with a naturally distributed structure. Although they commonly offer no (or little) quality guarantees, they were empirically found to produce high quality solutions [Yokoo and Hirayama, 1996; Maheswaran et al., 2004a; Zhang et al., 2005; Zivan et al., 2014].

The Distributed Breakout Algorithm (DBA) is one such local search algorithm, originally proposed to solve DisCSPs [Yokoo and Hirayama, 1996]. Agents in DBA associate weights with constraints and employ greedy local search to reduce the weighted sum of violated constraints. Agents try to escape local minima via a breakout mechanism that increases these weights.

DCOPs generalize DisCSPs by replacing hard constraints with cost functions that factor a global objective function. Initial attempts to adapt DBA to DCOPs focused on minimizing the number of constraint violations in over-constrained DisCSPs (also known as DisMaxCSPs) [Hirayama and Yokoo, 1997; Wittenburg and Zhang, 2003]. DBA was subsequently used with more general-valued DCOPs (i.e., problems in which a violated constraint incurs a numeric cost) [Zhang et al., 2005], but the details of this adaptation were omitted. This is unfortunate, because many details of DBA become unclear with general-valued cost functions. In DisCSP an assignment can either violate or satisfy a constraint. What does it mean for a general-valued DCOP constraint to be violated, when each combination of assignments incurs some cost? In DisCSP the weight of a constraint is increased by one. For a DCOP should this be interpreted as an increment to the constraint cost (e.g., increasing a base cost of 3 to 4, then to 5, etc.) or as an increment to a multiplicative factor of the constraint cost (e.g., increasing a base cost of 3 to 6, then to 9, etc.)? In DisCSP a constraint is a single joint assignment to a set of variables, and the scope of the weight increase is exactly that single joint assignment. In DCOP, constraints are arbitrary functions from joint assignments to costs, so what set of joint assignments should be affected by an increase?

In this paper we make three major contributions. First, we formalize the three design choices required to adapt DBA to general-valued DCOPs: the manner of computing effective costs based on true costs and modifiers (i.e., the “weights” of the original DBA), the definition of constraint violation, and the scope of changes to the modifiers during breakouts. For each choice we consider alternatives that are consistent with the original DBA: three definitions, two manners, and four scopes, resulting in a space of 24 variants parameterized along three dimensions. We propose the Generalized Distributed Breakout Algorithm (GDBA) to span this space.

Second, we establish theoretical properties of GDBA. We establish equivalences between some variants on important classes of problems. We also prove limitations of GDBA on general-valued DCOPs. Although DBA is complete and
sound on graph coloring DisCSPs with tree topologies, we prove that DCOPs defined on the same topologies may not be solved optimally by some variants of GDBA.

Our third contribution is an extensive empirical analysis of GDBA on three classes of DCOPs: random graphs with unstructured cost functions, weighted graph coloring, and meeting scheduling. Our results show that GDBA finds equal or better solutions than state-of-the-art competitors, in contrast to previous reported results [Zhang et al., 2005].

2 Distributed Satisfaction and Optimization

DisCSPs and DCOPs have agents \( A = \{ A_1, \ldots, A_m \} \); variables \( X = \{ X_1, \ldots, X_n \} \) where each variable uniquely belongs to (is held by) one of the agents; finite domains \( D = \{ D_1, \ldots, D_m \} \) where \( D_i \) specifies the values that \( X_i \) can take; and constraints describing relationships between variables. We make the standard assumption that each agent controls exactly one variable (\( n = m \)), and hence we use the terms “agent” and “variable” interchangeably. We further adopt a common assumption that all constraints are binary, involving only two agents. The undirected constraint graph has vertices \( X \) and an edge \((X_i, X_j)\) if and only if \( X_i \) and \( X_j \) are constrained. Two variables are neighbors if they are adjacent in the constraint graph, and \( N_i \) denotes the set of neighbors of \( X_i \). Each agent has exclusive control over the value assignment of its variable, and knowledge only of its neighbors and the constraints it has with its neighbors.

The set of constraints in a DisCSP is \( C \). Each constraint \( C_i \in C \) between \( X_{11} \) and \( X_{12} \) is a pair of values \( C_i \in D_{11} \times D_{12} \) representing a no-good, i.e., a disallowed combination of values for \( X_{11} \) and \( X_{12} \). If an assignment takes the values disallowed by a constraint, it is said to violate that constraint. The agents’ goal is to choose a full assignment \( x \in X \) in every iteration to minimize the sum of weights of violated constraints.

The set of constraints in a DCOP is \( F \). Unlike the no-good representation of DisCSP constraints, each DCOP constraint \( F_{ij} \in F \) is a function \( F_{ij} : D_{ij} \rightarrow \mathbb{R}_{\geq 0} \) mapping pairs of values of the constrained variables to costs. Constraints are symmetric so \( F_{ij} \) and \( F_{ji} \) are alternate names for the same constraint. The agents’ objective is to choose a full assignment \( x = (x_1, \ldots, x_n) \in D_1 \times \cdots \times D_n \) minimizing the total cost \( F(x) = \sum_{F_{ij} \in F} F_{ij}(x_i, x_j) \).

We refer to the DCOP binary constraints as two-dimensional tables with rows and columns indexed by values in the domains of the two constrained variables. We adopt the convention that when referring to a table \( F_{ij} \) from the perspective of a specific agent \( i \), the values in \( D_j \) always index the rows. Thus, the tables from the perspectives of \( i \) and \( j \) for a constraint \( F_{ij} \in F \) are transposes of each other.

We assume that the tables are not uniform (otherwise they can simply be ignored by the agents).

3 Distributed Breakout Algorithm (DBA)

DBA is a synchronous local search algorithm designed for DisCSPs. Agents in DBA collectively maintain a full assignment \( x \) with each agent keeping a current value for its variable. Each agent also maintains a weight, initially 1, for each constraint it is involved in. The agents greedily update \( x \) in every iteration to minimize the sum of weights of violated constraints. Each agent calculates the maximal improvement in the sum of weights that it can achieve by unilaterally changing its assignment, then coordinates with its neighbors to ensure that only the agent with the greatest improvement (subject to deterministic tie-breaking) in its neighborhood is allowed to change its value.

To escape local minima, DBA modifies its objective function by increasing some of the weights. Detecting true local minima requires the full assignment to be known to a single agent, so agents instead detect a weaker condition, quasi-local minima, that can be computed using only information about their neighbors.

Definition 1 \( A_i \) is in a \textit{quasi-local minimum (QLM)} if neither it nor any of its neighbors can unilaterally change to an assignment resulting in lower effective cost.

When an agent detects that it is in a QLM, it performs a \textit{breakout} by increasing the weights of all violated constraints, eventually causing it to choose a new value. Neighboring agents will not necessarily be in QLMs at the same time, and hence one neighbor may break out while the other does not. This may lead to agents having different weights for the same constraint. (A later revision to DBA disallowed this [Hirayama and Yokoo, 2005], but we follow the original description [Yokoo and Hirayama, 1996].)

4 Generalized DBA (GDBA)

DBA was originally designed for solving DisCSPs but the general structure of the algorithm is compatible with DCOPs. In fact, agents in DBA implicitly try to solve a DisMaxCSP, searching for an assignment with cost 0. GDBA (pseudocode

Algorithm 1: GDBA

1. Initialize cost modifiers to 0
2. Choose random value \( x_i \in D_i \)
3. Send \( x_i \) to all neighbors
4. while termination condition not met do
5. \( x_i' \leftarrow \arg \min_{x_i \in D_i} \sum_{j \in N_i} \text{EffCost}(d, j, x_j) \)
6. \( \Delta_i \leftarrow \sum_{j \in N_i} \text{EffCost}(x_i', j, x_j) - \text{EffCost}(x_i, j, x_j) \)
7. Send \( \Delta_i \) to all neighbors
8. Receive \( \Delta_j \) from all neighbors
9. if \( \Delta_i > 0 \) then
10. \( v_i \leftarrow x_i' \)
11. else if no neighbor can improve then
12. foreach \( j \in N_i \) do
13. if IsViolated\( (x_i, j, x_j) \) then
14. \( x_i \leftarrow \text{IncrementMod}(x_i, j, x_j) \)
15. Send \( x_i \) to all neighbors
16. Assign best \( x_i \) to \( X_i \)

Figure 1: Generalized DBA executed by \( A_i \).
in Figure 1) extends DBA to DCOPs with general constraint costs, where the original description may be interpreted in multiple ways, all of which are equivalent on DisMaxCSPs.

Each agent $A_i$ uses a modifier function $M_{ij} : D_i \times D_j \rightarrow \mathbb{Z}_{\geq 0}$ for each constraint $F_{ij}$, analogous to the weights in DBA. As with weights, these modifiers are increased during breakout. The modifiers combine with the base costs of the DCOP constraints to yield local effective cost functions (EffCost), which generalize the sum of weights in DBA. They are initialized to 0 in line 1 and increased by INCREASE_MOD. A synchronous step is defined by every agent receiving messages from all of its neighbors. Each iteration requires one step (lines 5 – 8) to calculate the best local improvement $\Delta_i$ based on the values received from neighbors, and another step (lines 9 – 17) to decide if $A_i$ can change its assignment. If $A_i$ detects a QLM, it performs the breakout process described in lines 14 – 16.

Detecting an optimal solution in a DCOP is generally intractable (unless P=NP) and so the search continues even if the optimal solution has been encountered. We assume an upper limit on the number of iterations as the termination condition (line 4), as it is simple and predictable. GDBA can also use a method like the Anytime Local Search framework [Zivan et al., 2014] to cache the best solution found during the search and assign it upon termination (line 18). Without such a method, the solution reported is the assignment held by the agents following the last iteration of the algorithm.

GDBA has the same time and space complexity of DBA. Each agent $A_i$ requires $O(|N_i| \cdot |D_i|)$ time in each step and $O(|N_i| \cdot |D_i| \cdot \max_{j \in N_i} |D_j|)$ space total. (Note that all agents execute each step in parallel.)

To complete the design of GDBA we next define the three subroutines EffCost, IsViolated, and INCREASE_MOD that implement the design alternatives summarized in Table 1.

**Manner of cost increase** defines how base costs and modifiers are combined to yield effective costs and is implemented in EffCost. We consider two manners. The first, $M$, uses modifiers as multiplicative factors to the base costs (line 20). This amplifies inherent differences in the base constraints, allowing for faster modification of the cost landscape while preserving some of the underlying problem structure. The second, $A$, uses modifiers as additive penalties to the base costs (line 21), allowing for finer-grained but slower modification of the cost landscape, and a risk of erasing inherent cost differences, resulting in an ill-fitting objective function.

### Table 1: Summary of the design alternatives for GDBA.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>Multiplicative manner.</td>
</tr>
<tr>
<td>$A$</td>
<td>Additive manner.</td>
</tr>
<tr>
<td>NZ</td>
<td>Non-zero constraint violation.</td>
</tr>
<tr>
<td>NM</td>
<td>Non-minimum constraint violation.</td>
</tr>
<tr>
<td>MX</td>
<td>Maximum constraint violation.</td>
</tr>
</tbody>
</table>

| E     | Entry scope.                 |
| C     | Column scope.                |
| R     | Row scope.                   |
| T     | Table scope.                 |

### Function EffCost($x_i, j, x_j$)

19 switch manner do
20 | case $M$: do return $F_{ij}(x_i, x_j) \cdot [M_{ij}(x_i, x_j) + 1]$
21 | case $A$: do return $F_{ij}(x_i, x_j) + M_{ij}(x_i, x_j)$

### Constraint violation is defined in IsViolated. We consider three possible definitions for constraint violations based on the base costs of the constraints. The use of base costs rather than effective costs is consistent with DBA, which only uses the presence, not the weight, of a no-good for determining constraint violation. It also helps to keep the cost landscape closer to the true objective, because only “bad” (i.e., violated) constraints are modified in breakouts. The three definitions are progressively more restrictive in defining violation, leading to less exploration but greater chance of convergence.

### Function IsViolated($x_i, j, x_j$)

22 $c \leftarrow \text{EffCost}(x_i, j, x_j)$
23 switch violation definition do
24 | case NZ: do return $c > 0$
25 | case NM: do return $c > \min_{d_1 \in D_i, d_2 \in D_j} F_{ij}(d_1, d_2)$
26 | case MX: do return $c = \max_{d_1 \in D_i, d_2 \in D_j} F_{ij}(d_1, d_2)$

A constraint is NZ-violated if it has a non-zero cost in an assignment (line 24). Some constraints may not have 0 cost for any assignment and hence will always be NZ-violated; although this will prevent convergence, it allows for more exploration that may result in better solutions. A constraint is NM-violated if its cost under the current assignment is non-minimum over all joint assignments (line 25), this is less explorative than NZ but more likely to converge because for every constraint there is always a joint assignment that does not NZ-violate it. Finally, a constraint is MX-violated if it takes the maximum cost over all possible joint assignments (line 26). Breakouts will only occur when an agent is trapped by its neighbors in its worst assignment; for other cases, GDBA with MX is not explorative and relies solely on greedy hill climbing for improvement. For every constraint there is always a joint assignment that does not MX-violate it.

### Scope of cost increase specifies the elements in $M$ that are incremented when the breakout is performed for a violated constraint. We consider four scopes named for the dimension in the table $M$ that is altered and implemented in INCREASE_MOD. In DBA, the weight of a single no-good is incremented. The analogue in GDBA is to increment the single entry corresponding violated no-good; we call this the $E$ scope. This is a very fine-grained modification that results in slower evolution of the cost landscape but more accurately captures the local minima, because an agent must actually be trapped in a QLM to modify the cost of a joint assignment. The $C$ scope affects the costs in a column: all values in the domain of $X_i$ given the neighbor’s current value. With $M$
manner this scales the relative importance of violated constraints for the agent’s current value; as the agent is trapped in its current value, this eventually allows it to escape. (We prove in Proposition 3 that it is not very interesting for A.) The R scope affects the costs in a row: all entries for the current value of X_i for all value assignments of the neighbor. This is similar to a value penalty in DisPeL [Basharu et al., 2005], trying to move the agent away from a value because it previously led to a QLM, even if the current context is different. Finally, the T scope affects the cost in the whole table: all entries in the modifier for a violated constraint are incremented, effectively making the constraint more important to A_i relative to its other constraints. T modifies the cost landscape the fastest and most drastically of all scopes.

\[ \text{Function \ INCREASEMOD}(x_1, j, x_j) \]

\[
27 \text{ switch scope do}
28 \text{ case E: do } \text{ Increment } C_{ij}(x_1, x_j)
29 \text{ case C: do for } d \in D_i \text{ do } \text{ Increment } C_{ij}(d, x_j)
30 \text{ case R: do for } d \in D_j \text{ do } \text{ Increment } C_{ij}(x_i, d)
31 \text{ case T: do for } d_1 \in D_i, d_2 \in D_j \text{ do } \text{ Increment } C_{ij}(d_1, d_2)
\]

5 Theoretical Results

We identify specific variants of GDBA with triples (manner, definition, scope) specifying the three dimensions. Because there are two manners, three violation definitions, and four scopes, there are a total of 24 GDBA variants. However, in this section we show that some of these may be equivalent for some classes of problems, i.e., in all iterations they compute the same effective costs and hence send the same messages and select the same assignments. When a dimension is restricted to a subset of the possible implementations, we represent it as a set in the triple notation. For example, \((M, \{NZ, NM\}, E)\) is DBA with \(M\) manner, \(E\) scope, and either \(NZ\) or \(NM\) constraint violation. If a dimension may take any value, we represent it with “*”. For example, \((*, NZ, E)\) means DBA with \(NZ\) constraint violation, \(E\) scope, and any manner.

5.1 Equivalences

We start with DCOPs with binary-valued constraints:

**Proposition 1** If for all \(F_{ij} \in \mathcal{F}\) the set \(\{F_{ij}(d_1, d_2) \mid d_1 \in D_i, d_2 \in D_j\} = \{0, 1\}\), then \((*, *, E)\) are equivalent.

**Proof:** The equivalence of the three violation definitions follows directly from the assumption that all constraints have minimum cost of 0 and maximum cost of 1.

Equivalence of manners is shown by induction on the number of iterations. We show that \(M_{ij}(d_1, d_2)\) is the same under both \(M\) and \(A\) for all \(M_{ij}\) and \(d_1 \in D_i, d_2 \in D_j\) and that the effective costs are the same. In the first iteration all \(M_{ij}\) are initialized as zero and so the effective cost is \(F_{ij}\) for both manners and the base case is proven.

Assume next that the \(M_{ij}\) and effective costs are the same for both manners on iteration \(t\). Thus the same value will be chosen for time \(t + 1\) and the same constraints will be broken out of at time \(t\) if \(A_i\) is in a QLM. Hence on iteration \(t + 1\), \(M_{ij}(x_i, x_j)\) will be the same for both manners. There are now two cases. If \(F_{ij}(x_i, x_j) = 1\) then the effective cost is \(M_{ij}(x_i, x_j) + 1\) under both manners. If \(F_{ij}(x_i, x_j) = 0\) then \(M_{ij}(x_i, x_j) = 0\) because the scope is \(E\) and \(F_{ij}(x_i, x_j)\) is not a violation. Thus under both manners the effective cost is 0 and equivalence is proven. \(\square\)

**Corollary 1** DBA is \((*, *, E)\) on DisCSPs and DisMaxCSPs where every constraint can be satisfied and violated.

**Proposition 2** With \(M\) manner and a fixed violation definition \(V \in \{NZ, NM, MX\}\), the variants \((M, V, \{E, C, R\})\) are equivalent on graph coloring problems.

**Proof:** In graph coloring, base costs \(F_{ij}(x_i, x_j) > 0\) if and only if \(x_i = x_j\), and so because of the \(M\) manner, the effective cost will also be non-zero if and only if \(x_i = x_j\). This means that modifiers are irrelevant except for \(M_{ij}(d_1, d_2)\) for \(d_1 \in D_i\), Constraint \(F_{ij}\) can only be violated (under any definition) when \(x_i = x_j\), and for the scopes \(E, C, R\), \(M_{ij}(d_1, d_2)\) is only incremented when \(x_i = x_j = d_1\). Thus they are all equivalent.

Note that \((M, V, T)\) is not equivalent to these, since for \(d_i \in D_i\), the modifier \(M_{ij}(d_i, d_i)\) can also be incremented when \(x_1 = x_2 = d_2 \neq d_i\). \(\square\)

The next proposition shows that breakouts do not change the behavior of agents in some variants of GDBA.

**Proposition 3** \((A, *, \{C, T\})\) is equivalent to MGM, i.e., as if no breakouts are performed at all.

**Proof:** We consider a variable \(X_i\) that is in a QLM and show that it does not change the choice of assignment after breakout. Let \(J\) be the set of \(j\) such that \(F_{ij}\) are currently violated, so that the breakout increments \(M_{ij}(x_i, x_j)\) for all \(j \in J\). Denote the effective cost before and after breakout as \(\text{EffCost}_i\) and \(\text{EffCost}_{i+1}\), respectively. Then note that because we are in \(C\) or \(T\) scope,

\[
\text{EffCost}_{i+1}(d, j, x_j) = \begin{cases} 
\text{EffCost}_i(d, j, x_j) + 1 & j \in J \\
\text{EffCost}_i(d, j, x_j) & \text{otherwise}
\end{cases}
\]

Line 6 of Algorithm 1 computes the best alternative after
breaking preferences on different types of problems) from 0.1 to 0.9 were used and these performed best, each algorithms: DSA (type C, with averages over 200 independent, randomly-generated instances

6 Experimental Results

The results presented in Figure 3(b) demonstrate that breakouts do not always lead to better solutions. In particular, the poor performance of \( R \) scope is explained by breakouts causing inferior solutions to be explored, which also occurred with \( E \) scope with \( NM \) or \( NZ \) definitions.

\( E \) scope was most effective when paired with \( MX \) definition, particularly in the \((A, MX, E)\) configuration. Although this led to worse solutions being explored on average (as seen by current cost increasing with step), it also allowed GDBA to continue to occasionally find better solutions (as seen by anytime costs continuing to slowly decrease). This used a relatively slow rate of exploration, since \( MX \) definition is the most restrictive violation definition (and hence the fewest assignments lead to breakouts) and \( E \) is the most restrictive scope

Figure 3: Anytime cost (a) and current cost (b) of GDBA on unstructured problems.
(and so each breakout results in the smallest modification to the cost landscape). For $T$ scope, exploration through breakouts was really only useful in the $(M, NM, T)$ configuration, as can be seen by the lower anytime costs vs. the current costs, but this balance between exploration and exploitation was the most successful overall.

Similar trends were observed in the graphs presenting the results on weighted graph coloring and meeting-scheduling. The figures were omitted for lack of space. The $(M, NM, T)$ configuration dominated over all benchmarks.

Figure 4 plots the average anytime costs of $(M, NM, T)$ and the competing algorithms at each step of execution on random unstructured problems. All differences at step 2000 are statistically significant for $p$-value $< 0.01$. DSA $p = 0.8$ performs the best out of all competing algorithms, quickly finding solutions of better quality on average than those found by the others; with $p = 0.4$ it finds considerably worse solutions. $(M, NM, T)$ initially improves more slowly than DSA, which is not surprising considering that agents are only allowed to change values on every other step, while agents in DSA may change their values on every step if doing so leads to a local improvement. Despite this initial relative slowness, within 250 steps $(M, NM, T)$ finds better solutions than all competing algorithms other than DSA, and within 500 steps it surpasses even DSA and continues to improve slowly (on average) when given more time.

We next considered coloring random graph topologies with $p_1 = 0.05$, $d = 3$, and random costs on $[1..10]$. Figure 5 presents the average anytime costs. Both DSA with $p = 0.8$ and DSAN very quickly find very good solutions (whose costs are not significantly different, although DSA finds its solutions slightly faster. DSA with $p = 0.4$ again finds much worse solutions. $(M, NM, T)$ again takes longer to improve its solutions, but within 750 steps it finds solutions that are statistically better than all the competitors, including DSA. With more time, it continues to improve.

Our final set of experiments were on meeting scheduling problems using EAV representation [Maheswaran et al., 2004b] with 200 people and 50 meetings to be scheduled in a 10-hour workday from 8:00 AM to 6:00 PM discretized into 15-minute time slots. Each meeting has a duration uniformly distributed on $[1..4]$ time slots, and a desired attendance uniformly distributed on $[2..6]$ people; the specific individuals are chosen uniformly at random. The required traveling time between the locations of each pair of meetings is uniformly distributed on $[1..2]$. The preferences of the participants impose a cost that is uniformly distributed on $[1..3]$ for each meeting and time. Two meetings scheduled so that common participants do not have enough time to attend both impose a conflict cost equal to the sum of the time slots missed by those participants in both meetings.

Figure 6 average anytime costs. Fast convergence suggests that breakouts are not effective in this setting, and that the bulk of $(M, NM, T)$’s improvement is due to greedy local search. Thus, it is not surprising that it behaves very similarly to MGM, which is just the local search component of DBA without a breakout mechanism. However, breakouts do confer a very small advantage to $(M, NM, T)$, which finds better solutions than all other algorithms except DSAN and DSA (with $p = 0.4$), at a statistically significant level of $p$-value $< 0.01$; DSAN and DSA find solutions of exactly equal quality, albeit more slowly. DSA with the higher value of $p = 0.8$ did poorly on these problems, demonstrating that good DSA performance relies heavily on problem-specific parameter tuning; in contrast, $(M, NM, T)$ performs well on different problems without needing such tuning.

7 Conclusion and Future Work

DBA is a local search algorithm designed to solve DisCSPs and DisMaxCSPs. Following early work comparing DSA and DBA, the common assumption of researchers was that DSA is superior, and therefore DSA was used for comparison in recent studies of incomplete DCOP algorithms.
We demonstrate in this paper that the performance of DBA when solving general-valued DCOPs is dependent on the design choices made. GDBA, generalizes DBA by allowing 24 combinations of 3 design choices. Our results demonstrate that GDBA, specifically the \((M, NM, T)\) variant, is an effective algorithm for a wide range of general-valued DCOPs. This contrasts with DBA’s well-known limitations on DisCSPs and DisMaxCSPs. Many other DCOP algorithms that originated as DisCSP algorithms may benefit from a similar rigorous study. For example, DSA-A and DSA-B differ in their behavior only when there are violated constraints; this invites investigation of violation definitions for DSA. (We used DSA-C, which does not require such definition.)

DBA, as a distributed implementation of a centralized method [Morris, 1993], is a bridge between DisCSPs and centralized CSPs. Future work should explore what other centralized approaches may be adapted to the distinct but related DisCSP and DCOP fields.

References


