Escaping Local Optima with Penalties in Distributed Iterative Improvement Search

Muhammed Basharu, Ines Arana, and Hatem Ahriz

School of Computing, The Robert Gordon University,
Aberdeen, AB25 1HG, United Kingdom.
{mb, ia, ha}@comp.rgu.ac.uk

Abstract. The advantages offered by iterative improvement search make it a popular technique for solving problems in centralised settings. However, the key challenge with this approach is finding effective strategies for dealing with local optima. Such strategies must push the algorithm away from the plateaux in the objective landscape and prevent it from returning to those areas. A wide variety of strategies have been proposed for centralised algorithms, while the two main strategies in distributed iterative improvement remain constraint weighting and stochastic escape. In this paper, we discuss the two phased strategy employed in Distributed Penalty Driven Search (DisPeL) an iterative improvement algorithm for solving Distributed Constraint Satisfaction problems. In the first phase of the strategy, agents try to force the search out of the local optima by perturbing their neighbourhoods; and use penalties, in the second phase, to guide the search away from plateaux if perturbation does not work. We discuss the heuristics that make up the strategy and provide empirical justification for their inclusion. We also present some empirical results using random non-binary problems to demonstrate the effectiveness of the strategy.

1 Introduction

Recent advances in communications technology have created new types of problems that require systems of autonomous agents to interact and negotiate for possible solutions. Some of these problems, such as scheduling and resource allocation, can be formalised as Distributed Constraint Satisfaction Problems (DisCSPs) where they are decomposed into variables and constraints partitioned amongst the agents involved. DisCSPs are solved by a collaborative search process in which agents try to find valid combinations of values that satisfy all the constraints.

Most of the search algorithms for solving DisCSPs are based on systematic backtracking, where agents take turns, either synchronously or asynchronously, to select values for the variables they represent or to detect invalid selections made by other agents. While these algorithms have been shown to be complete, they still inherit some of the drawbacks of backtracking in general. Iterative improvement search for distributed constraint reasoning was introduced in the form of the Distributed Breakout Algorithm (DBA) [16] and the Distributed Stochastic Algorithm (DSA) [17] as alternatives to backtracking. Although these algorithms are incomplete, they come with the advantage of being able to converge quicker on large problems than backtracking based approaches. However they can also converge quickly to local optima. To deal with this, weights are attached to constraints in DBA and those on violated constraints are increased whenever the search is stuck. While in DSA, agents try to avoid local optima by stochastically deciding when to make hill climbing moves.

In this paper, we discuss a two phased strategy for dealing with local optima in the Distributed Penalty Driven Search (DisPeL) algorithm. In the first phase, agents try to perturb their neighbourhoods aiming to push the search out of the current plateau it occupies. And in the second phase, agents increase penalties attached to domain values to in order to reshape the objective landscape and guide the search away from the plateau. We discuss the impact of these heuristics and present results of empirical evaluations carried out using random non-binary DisCSPs.

The remainder of this paper is structured as follows. We start with brief preliminaries on distributed constraint satisfaction problems in Section 2. After which, the algorithm and its strategy is discussed. Following that we present results of the empirical evaluations. In Section 5 we discuss the contributions of the various parts of the strategy and discuss related work in Section 6.

2 Background

A Constraint Satisfaction Problem (CSP) is formally defined as a triple (V, D, C) comprising a set of variables (V), a set of domains (D) listing possible values that may be assigned to each variable, and a set of constraints (C) on values that may be simultaneously assigned to the variables. The solution to a CSP is a complete assignment of values to variables satisfying all the constraints.

DisCSPs formalise problems that occur in distributed environments within the CSP framework [15]. In these scenarios, information about a problem is held by a coalition of participants and for some reasons (such as security or privacy) the information can not be collected centrally in one location. Each participant in the DisCSP is therefore represented by an agent, which is aware of all the participant's variables[1] and the constraints they are involved in. Agents collaborate to solve the DisCSP, each seeking to find assignments for its variables that satisfy all attached constraints.

3 Distributed Penalty Driven Search

3.1 Overview

Distributed Penalty Driven Search (DisPeL) was first introduced in [3] as an adaptation of the centralised Guided Local Search [12] for solving DisCSPs. In [4], we presented a new version of the algorithm and changed the name to reflect the nature of its strategy. Both versions used similar strategies for dealing with local optima, but differ fundamentally in how the growth of penalties are controlled. In the first version, an upper bound is placed on the size of penalties. While, in DisPeL all penalties are discarded periodically.

[1] Here we assume that each agent represents just one variable.

DisPeL is a synchronous iterative improvement algorithm for solving DisCSPs. It is essentially a greedy algorithm that starts with a random initialisation, which agents try to improve by selecting values that minimise the number of constraints violated in each iteration. To deal with local optima, a two phased strategy is employed as follows: (i) perturbation phase: it tries to perturb the local neighbourhood to force agents to try other combinations of values (i.e. to explore other areas of the search) and; (ii) learning phase: if perturbation does not resolve the deadlock, it tries to learn about and avoid the value combination that caused the deadlock.

Perturbation as way of dealing with local optima is fairly common with centralised local search algorithms, and typically comes in the form of stochastic actions aimed at pushing an algorithm out of the plateau it occupies. In DisPeL's perturbation phase, we use a temporary penalty to try and force agents to consider combinations of values other than the current one. The temporary penalty was selected as a result of experiments we conducted where a greedy algorithm was pushed to a local optimum, various perturbation mechanisms applied to it and their effects evaluated. The temporary penalty came out strongest because it did not create as many new violations in other parts of the problem as the other alternatives despite not resolving as many of the original violations as some other mechanisms.

In the learning phase (second part of the strategy), DisPeL tries to learn about and avoid bad assignments i.e. those associated with local optima. Incremental penalties are attached to each domain value and incorporated into the objective function. When a perturbation is unable to resolve a conflict, the incremental penalties attached to all values associated with the conflict are increased. The desired effect is twofold. First, it changes the shape of the objective landscape making surrounding areas more promising, and secondly, it makes agents less likely to select those values as the search progresses unless they offer significant improvements to the objective function.

To tie both parts of the strategy together, we use a no-good store to keep track of a fixed number of recent conflicts. Therefore whenever the temporary penalty is used, the assignments that make up the no-good causing the conflict is placed in the store. As such, when next there is a deadlock an agent can find out if a previous attempt at resolving it has been made, and hence decide on the appropriate course of action.

3.2 Algorithm details

The objective function (h) for each agent is defined as follows:

$$h(d_i) = v(d_i) + p(d_i) + \begin{cases} t & \text{if a temporary penalty is imposed} \\ \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where:

d_i is the ith value in the variables domain
$v(d_i)$ is the number of constraints violated if d_i is selected
$p(d_i)$ is the incremental penalty attached to d_i
t is the temporary penalty ($t > 1$)

The temporary penalty is used in a single iteration and it has to be discarded immediately after it is used. The temporary penalty can be any integer greater than 1, and its size does affect the overall behaviour of the algorithm. With a small temporary penalty (e.g. $t = 2$) it is possible that some agents are not forced to change the values of their variables because the alternatives are significantly worse than the deadlock state. As such, the perturbation to the neighbourhood may not travel far beyond some agents. With a large temporary penalty (e.g t = 100), all agents imposing temporary penalties are forced to change their variables' values and the perturbation is likely to percolate further away in the constraint graph from the agent that initiated it. While this may be beneficial on some types of problems, it also has detrimental effects on many types of problems. We use $t = 3$ in all experiments reported in this paper, irrespective of the problem size. We discuss the impact of the temporary penalty further in Section 5.

Incremental penalties attached to values associated to a deadlock are increased when the perturbation fails to resolve it. While this allows agents to avoid bad assignments, there is potential for the incremental penalties to dominate the objective functions to the extent that it possibly diverts the search away from promising regions. To deal with this, we reset the incremental penalties to zero: (i) when agents find consistent values for their variables and (ii) periodically. In the former, simply because it is assumed that the penalties have become redundant. While in the latter, the penalties are reset to keep potential paths to solutions open. This is somewhat risky because if penalties are reset too often, search experience is lost too quickly and there is not much benefit of using the penalties in the first place. While resetting penalties after long periods can affect the objective function such that rather than seeking to minimise the number of constraints violated, emphasis shifts to minimising the penalties. The alternative to resetting penalties is to allow them to decay periodically, as done in [9], so that search memory is not entirely lost every so often. For the periodic resets, we have been able to establish from empirical experiments that performance is optimal (especially in terms of search cost) if it is done every six iterations. This value is used for all experiments (including those reported here), irrespective of problem size, type, or structure.

Each agent has a no-good store to help determine appropriate actions for deadlock resolution, with which it maintains a list of recent no-goods on a First-In-First-Out basis. A no-good is an agent's AgentView comprising all its neighbours' current assignments. No-goods are specifically used as short term memory and are not considered as new constraints and, therefore the number stored is limited in order to save memory. As a rule of thumb, we fix the size of the no-good store for each agent to *N*; where *N* is the number of neighbours the agent has. Specifically to take into account the size of the individual DisCSP being solved [2]. The size of the no-good store can also determine how often agents perturb their neighbourhoods, affecting the overall efficiency of the algorithm. Too many perturbations can cause the algorithm to wander about in the search space reducing exploitation activity, while a large no-good store cuts down on the necessary exploration activity.

[2] This also helps us keep our comparisons with other algorithms fair, since we are not optimising it for each problem type or size.

The pseudo-code of the algorithm is outlined in Figures 1, 2, and 3.

3.3 Agent behaviour

At initialisation, agents create a static ordering using part of the Distributed Agent Ordering algorithm [6] so that unconnected agents can act in parallel. Agents do this individually by partitioning their neighbours into a set of higher priority (those with lower IDs) and lower priority (those with higher IDs) neighbours. During the search, agents will communicate with both sets of neighbours but would only become active (i.e. to select values for their variables) after receiving messages from all higher priority neighbours.

In the normal course of the search, an agent selects a value that minimises equation (1) and informs its neighbours of this value. If the agent is stuck at a quasi-local-minimum, it initiates the conflict resolution process as described earlier. We define a quasi-local-minimum as a situation where the AgentView of an agent with an inconsistent variable is unchanged in two consecutive iterations (Figure 2, line 2). Given that the agent will always select the value minimising the number of constraints violated, if its neighbour's values are unchanged from one iteration to the next, then it obviously means that there is no improvement forthcoming. This differs from the definition in [16].

```
initialise
do
    when active
        rpCounter++
        if rpCounter = 6
            reset incremental penalties
            rpCounter = 0
        end if
        if penalty message received
            respond_to_message()
        else
            if current value is consistent
                reset incremental penalties
                send message(id, value, null) to neighbours
            else
                resolve_conflict()
            end if
        end if
    return to inactive state
until terminate
```

Fig. 1. DisPeL: Agent main loop

To perturb its neighbourhood, a deadlocked agent imposes a temporary penalty on the current value of its variable and at the same time, requests all lower priority agents

```
 1 procedure resolve_conflict()
 2     if agentView(t) $\neq$ agentView(t-1)
 3         select value minimising objective function
 4         send message(id, value, null)
 5         return
 6     end if
 7     if agentView(t) is not in no-good store
 8         add agentView(t) to no-good store
 9         impose temporary penalty on current value
10         select value minimising objective function
11         send message(id, value, addTempPenalty)
12     else
13         increase incremental penalty on current value
14         select value minimising objective function
15         send message(id, value, increasePenalty)
16     end if
17 end procedure
```

Fig. 2. DisPeL: Initiating the conflict resolution process.

```
 1 procedure respond_to_message()
 2     if message is increase incremental penalty
 3         increase incremental penalty on current value
 4         select value minimising objective function
 5     else
 6         impose temporary penalty on current value
 7         select value minimising objective function
 8     end if
 9     send message(id, value, null)
10 end procedure
```

Fig. 3. DisPeL: Responding to a penalty message received from a higher priority agent

with variables violating constraints with its variable to do the same (Figure 2, lines 7-11). After which, it places the current AgentView in the no-good store for future reference. If later in the search the agent returns to the same deadlock (evident by its presence in the no-good store), it increases the incremental penalty attached to the current value of its variable and requests that all lower priority neighbours do the same (Figure 2, lines 13-15).

An agent receiving a penalty request cannot itself initiate conflict resolution, as it has become part of an ongoing process (Figure 1, lines 9-10). And for the obvious reason that a higher priority neighbour involved in the deadlock initiated the process. However, there may be times when an agent's variable is involved in more than one deadlock especially with multiple unconnected higher priority neighbours; and the agent is likely to receive conflicting penalty requests from those neighbours. In such a case, the request to impose a temporary penalty is ignored in favour of the increase in incremental penalties. In any case, if an agent receives multiple messages from different agents to do the same thing, it treats these messages as a single message. For example, it will not increase incremental penalties more than once in a single iteration even if it receives messages to do so from several agents.

4 Empirical Evaluation

We evaluate DisPeL's performance on random non-binary DisCSPs on two criteria: (1) number of problems solved and (2) the number of cycles (or iterations) taken to find the solutions. Using the number of cycles as measure of efficiency is justified by the fact that it is considered to be an independent metric that abstracts out effects like implementation and computing environment that can influence other metrics like CPU time [1]. Furthermore, in the case of synchronous algorithms, the cycle count can be used to directly infer other costs such as the number of messages exchanged between agents.

Random DisCSPs were generated using the standard Model B [11] modified as follows. First, support tuples were included in each constraint so that each problem is guaranteed to have at least one solution. And, secondly, constraints were randomly assigned to variables with preferential attachment [2, 14], so that the instances resemble real life problems i.e. the distribution of constraints to variables follow a power law. In the following, we summarise results of experiments evaluating the performance of the algorithm on different sizes and include a Run Length Distribution [8] analysis showing the variability in performance on a single instance.

In addition, we used DBA as the benchmark for comparing results. DSA was not included in the evaluations, even though it has been shown to outperform DBA on distributed scan scheduling problems. It converges quicker than DBA to local optima. But, Hirayama and Yokoo [7] showed that DSA rarely finds a solution in decision problems where the goal is to satisfy all constraints and explain that it remains stuck at local optima because there is no explicit mechanism for escaping deadlocks. As we are specifically interested in decision problems, we believe that it is not suitable to include DSA in the evaluations.

4.1 Variability on a single instance

The first results presented here show the empirical behaviour, using a Run Length Distribution plot, of both algorithms on a single problem instance as affected by their initial random instantiations. We use a non-binary DisCSP with 60 variables and a mixture of non-binary constraints with different arities (80 3-ary, 40 4-ary, and 20 5-ary constraints). Constraint tightness is fixed at 50% for all constraints and there are 10 values in each variable's domain. 500 attempts were made by each algorithm with a maximum cut-off of 10,000 iterations for DisPeL and 20,000 iterations for DBA[3].

The cumulative distributions plotted in Figure 4 suggest that DisPeL and DBA are quite sensitive to the initial random values selected for variables. Although DisPeL has a higher variability - its percentile ratio $Q_{0.75}/Q_{0.25}$ is 4.61 compared to 4.26 for DBA

[3] This is because agents change variable values once in every two cycles in DBA (i.e. the *wait_ok* and *improve?* cycles) compared to changing values every iteration in DisPeL

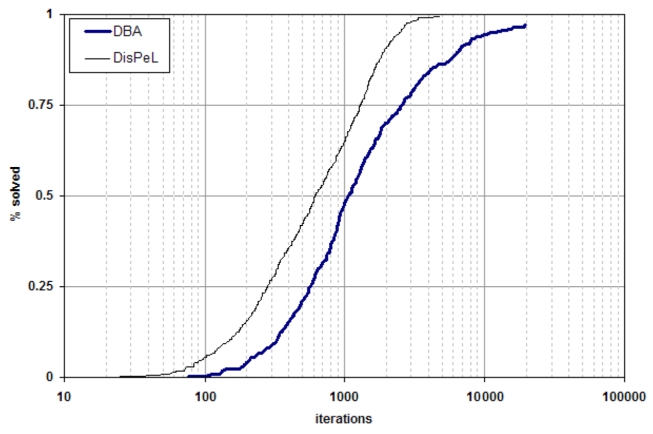
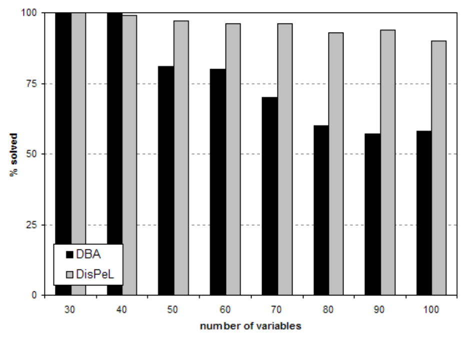


Fig. 4. Empirical Run Length Distributions of DisPeL and DBA on a single problem instance.

- it does have a higher probability of finding solutions and indeed had a higher success rate than DBA. The sensitivity to random instantiations suggests that both algorithms can benefit from a strategy of randomised restarts if optimal cut-offs can be determined.

4.2 Performance on different problem sizes

In further experiments, we compared both algorithms on a set of ternary problems with particular interest on the growth in search costs as the problem size (*n*) increases. The ratio of constraints to variables is held constant at 2:1, constraint tightness fixed at 0.55, and domain size is 10 (for each variable). For each problem size, we used 100 problems and limited DisPeL to $100n$ iterations and DBA to $200n$ iterations. The results of these experiments are plotted in Figures 5 and 6 which respectively show the percentage of problems solved, the average number of cycles required, the median cycles, and some quartiles ($Q_{0.25}$ and $Q_{0.75}$).



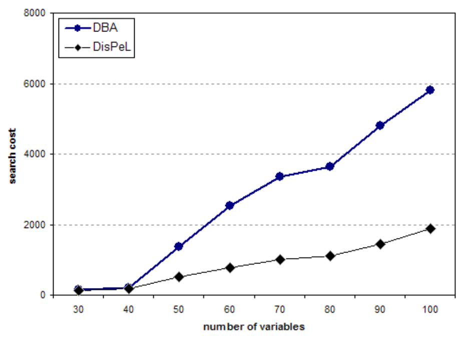


Fig. 5. Number of problems solved (left) and the average costs (right).

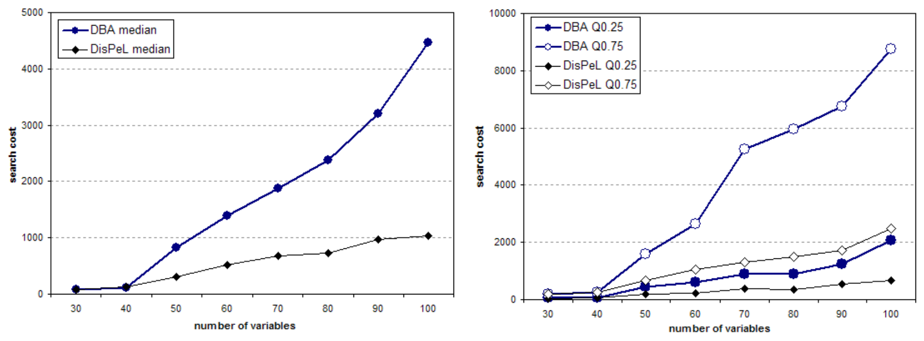


Fig. 6. Median search costs and quartiles for solving the problems in Figure 5.

Figure 5 shows that as the problem size increases, DBA solved fewer problems and the average search costs increased at a much faster rate than those for DisPeL. In Figure 6, the plot on the left shows that the median cost for DBA is higher than DisPeL's and the plots of the quartiles show that there is a much wider distribution of search costs for DBA than DisPeL.

4.3 Effect of constraint density

Finally, we report results of experiments studying how algorithm behaviour is affected by the constraint density. Results of the evaluations are summarised in the plots of Figures 7 and 8. 100 4-ary problems are generated for each point in the plots. There are 40 variables in each problem, 8 values in each variables' domain, and constraint tightness is fixed at 40%. The plots show a progression from sparse to dense problems, where the number of contraints (shown on the *x* axes) are steadily increased. We limited DisPeL to 10,000 cycles and DBA to 20,000 cycles on each attempt.

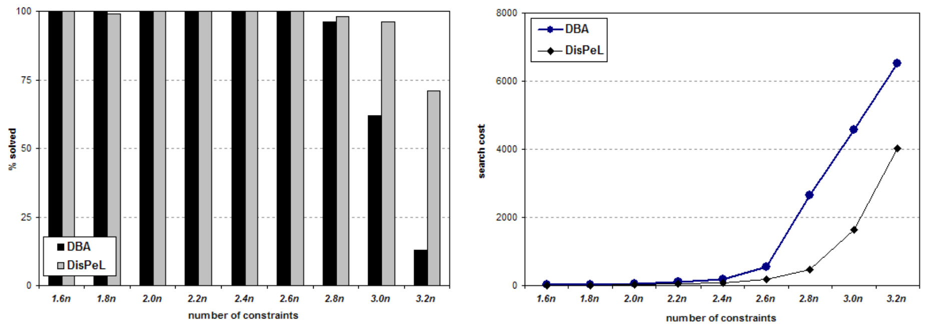


Fig. 7. Number of problems solved (left) and the average costs (right).

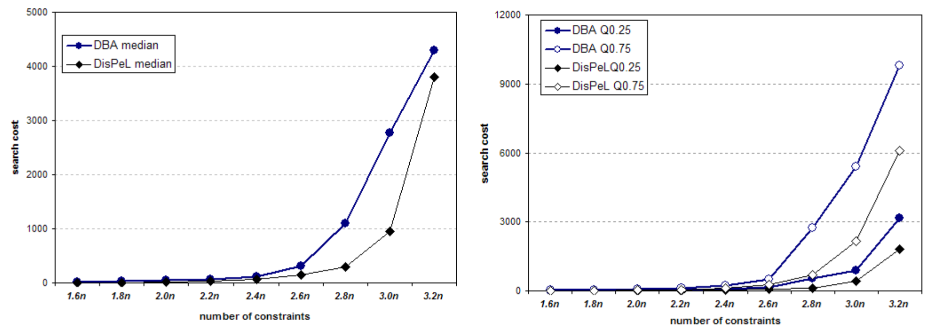


Fig. 8. Median search costs and quartiles for solving the problems in Figure 7.

Figures 7 and 8, show that both algorithms have identical performance on sparse problems, finding the same number of solutions and using about the same number of iterations to solve the problems. It gets interesting as constraint density increases, there is an abrupt drop off in the number of problems solved and an accompanying steep rise in search costs at the $3.0n$ mark for DBA. The effect on the number of problems solved is not as pronounced with DisPeL and there is a less dramatic increase in search costs from that point.

5 Discussions

5.1 Effect of resetting incremental penalties

In DisPeL agents reset all incremental penalties whenever they find consistent values for their variables and periodically. We argued that penalties become redundant when consistent values are found, and obscure the objective landscape if they are retained for too long. Empirical justification for these decisions is provided with results form an experiment comparing DisPeL, with a version of it where penalties are only reset periodically, and another version where penalties are reset only when consistent values are found. A version without any penalty resets was also tested. 50 problems were used, each with 40 variables, $2.3n$ constraints, and constraint tightness set to 0.55. All versions were started with the same initial values to rule out any random effects on the evaluation and limited to 4,000 iterations on each attempt. The results of the experiment are summarised in Table 1.

As expected, there is a massive performance gain from resetting penalties (at least within the DisPeL framework). While any form of resets [4] is beneficial, the combination of both reset strategies appears to be the best approach for the algorithm.

[4] Assuming periodic resets are not done too often or or not often enough.

Reset strategy	number solved	average cost
No resets	2	2,769
Resetting only when consistent values are found	45	1,611
Periodic resets alone	46	761
DisPeL	49	675

Table 1. Evaluating the effects of alternative reset strategies on 50 random problems.

5.2 Impact of the temporary penalty

The temporary penalty is used to perturb a neighbourhood when a conflict is first encountered. Giving agents opportunities to resolve some conflicts immediately, that would otherwise take a build up of incremental penalties to fix. Results from preliminary work, showed that when the temporary penalty was used to perturb a greedy algorithm at a local optimum, 57% of the original constraints violated where resolved in ensuing iterations. While, new constraint violations where caused in other parts of the constraint graph 43% of the time. In constrast to the incremental penalty, which resolved 65% of the violations but caused more constraints to be violated 9 out of every 10 times it was used [5].

t	number solved	average cost
2	97	178
3	100	172
4	99	173
5	99	280
6	99	249
7	99	299
8	100	249
9	100	286
10	100	270
15	100	287
50	100	287
100	100	287

Table 2. Evaluating the impact of the temporary penalty (t) size using 100 problems (60 variables, 120 constraints, tightness is 50%, and domain size is 8).

The size of the temporary penalty also affects the behaviour of the algorithm, determining how far perturbations percolate the network and the subsequent likelihood of a deadlock being resolved quickly. Table 2 summarises an empirical evaluation of the impact of the temporary penalty (t) size on performance. The results show that there is

[5] Looking at the immediate impact of the penalties and not considering the long term effect of accumulated penalties.

little difference in the results for temporary penalty values between 2 and 4, and average search costs suddenly increases with a value of 5 and remain at least 40% higher with higher values for *t*. Behaviour of the algorithm was identical for all runs with a temporary penalty value of 15 and above.

5.3 Effect of the no-good store size

No-goods are retained by agents to keep track of recent conflicts, and to help them decide what heuristic to use when conflicts are encountered. Because these no-goods are not taken as new constraints, only a limited number of them are held at any point in time. We do not specify set limits, because the size of the store has to change with the size of the problem being solved. If too few no-goods are held, there are going to be too many perturbations if agents regularly return to conflict states after long intervals. On the other hand, there is a point after which storing additional no-goods just uses up more memory and does not offer any improvements. Having said this, we limit the maximum number of no-goods held by agents to the number of neighbours (N) they have individually; although it may be optimised for an individual problem. We show that this upper bound is appropriate with the RLD in Figure 9, comparing it with an upper bound of $4N$. A DisCSP with 75 variables and 150 3-ary constraints is used for the experiment. There are 8 values in each variables domain and constraint tightness is fixed at 50%. The figure suggests that there is no performance gain from retaining too many no-goods. The explanation for this is that deadlock states, after being resolved, are not revisited too often. Therefore, agents need not retain a long history of their experiences.

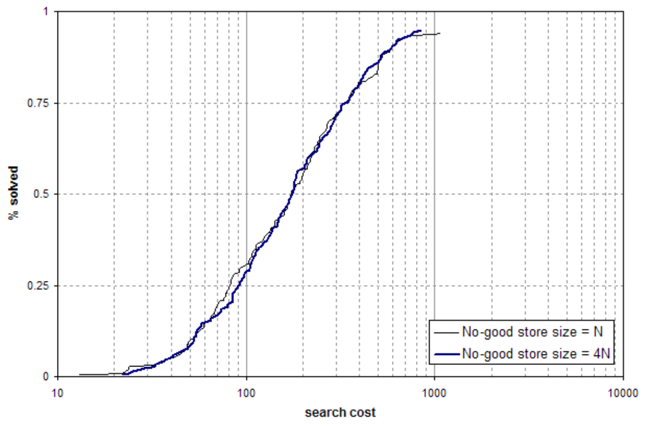


Fig. 9. Run Length Distributions showing the effect of the no-good store size on performance.

6 Related work

Several forms of penalty driven search have been developed in the literature especially for dealing with local optima in centralised hill-climbing algorithms. In algorithms like those presented in [13] and [5], penalties are attached to constraints and those on violated constraints are modified whenever the underlying hill-climbing search is stuck. These have the effect of modifying the objective landscape such that emphasis is placed on satisfying constraints regularly violated. The Breakout Algorithm [10] is similar to the aforementioned and it motivated the work on DBA, which extended and introduced this form of resolution for distributed constraint reasoning.

In a slightly different approach, penalties have been attached to problem features rather than the constraints in the Guided Local Search algorithm [12] with the same aim of contorting plateaux in the objective landscape. But the choice of features is often problem dependent. For example, non-overlapping blocks of domains are selected as features when solving non-convex optimisation problems. While for solving boolean satisfiability problems, clauses were selected as features in an extension of the algorithm [9]. GLS bears the closest resemblance to our work but there are major differences in the way penalties are incorporated into the objective function. In GLS, penalties are multiplied by a lambda parameter which moderates the impact on the objective function; and it is also used to control the exploration/exploitation behaviour of the search. Furthermore, the utility of penalising a feature is also estimated so that those features with higher costs are penalised first and the likelihood of a feature being penalised decreases the more times it is penalised.

Periodic penalty resets have also been considered in the literature. In [10], it was pointed out that accumulated weight increases (or penalty increases in this case) may conspire to block paths to a solution in the objective landscape. And, as such, restricting the algorithm to a sub-optimal region of the landscape which may result in infinite oscillations. Periodic resets were also used in a variant of GLS for solving the Quadratic Assignment Problem (QAP) in [12]. Based on an argument that it allows the search revisit solutions that include features penalised earlier, leading to an intensification of the search in profitable areas of the search space. But, it was also pointed out that the drawback of doing this is that the algorithm loses some of the exploration ability that pushes it towards unexplored areas of the search space. Results from that work showed that the reset strategy improved over the basic GLS with a higher percentage of successful runs. However, the mean quality of solutions was lower.

7 Summary

We have presented a distributed iterative improvement algorithm (DisPeL) for solving DisCSPs that relies on a two part penalty-based strategy for dealing with local optima. In the first part of the strategy, agents try to resolve local-optima by perturbing their neighbourhoods using a temporary penalty. And resort to the second part of the strategy if the deadlock is unresolved; where incremental penalties attached to domain values are increased to help agents avoid assignments linked to the deadlock. Both parts of the strategy are tied together by no-good stores maintained by each agent, which keep track of recent conflicts. The component parts of the strategy were discussed, and empirical justification for their inclusion was also provided. Collectively, the heuristics show that retention of too much search memory hinders the algorithm's performance. The algorithm was evaluated using random non-binary DisCSPs and its performance was compared against DBA. The results show thatDisPeL consistently solved more problems than DBA and it required fewer iterations to solve the problems.

References

1. Ravinda K. Ahuja and James B. Orlin. Use of representative operation counts in computational testing of algorithms. *INFORMS Journal of Computing*, 8(3):318–330, June 1996.
2. Albert-Lszl Barabsi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003.
3. Muhammed Basharu, Ines Arana, and Hatem Ahriz. Distributed guided local search for solving binary DisCSPs. In Ingrid Russell and Zdravko Markov, editors, *Proceedings of the 18th International FLAIRS Conference (FLAIRS 2005)*, pages 660–665. AAAI Press, May 2005.
4. Muhammed Basharu, Ines Arana, and Hatem Ahriz. Solving DisCSPs with penalty driven search. In *Proceedings of 20th National Conference on Artificial Intelligence (AAAI-05) - To appear*. AAAI Press, 2005.
5. P. Galinier and J. Hao. A general approach for constraint solving by local search. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'00)*, March 2000.
6. Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11 (2):167–188, June 2002.
7. Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. *Artificial Intelligence*, 161(1–2):89–115, January 2005.
8. Holger H. Hoos. *Stochastic Local Search - methods, models, applications*. PhD thesis, Darmstadt University of Technology, Germany, 1998.
9. Patrick Mills and Edward Tsang. Guided local search applied to the satisfiability (SAT) problem. In *Proceedings of the 15th National Conference of the Australian Society for Operations Research (ASOR'99)*, pages 872–883, July 1999.
10. Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 40–45, 1995.
11. Edgar M. Palmer. *Graphical evolution: an introduction to the theory of random graphs*. John Wiley & Sons, Inc., 1985.
12. Christos Voudouris. *Guided local search for combinatorial optimisation problems*. PhD thesis, University of Essex, Colchester, UK, July 1997.
13. Benjamin W. Wah and Zhe Wu. The theory of discrete lagrange multipliers for nonlinear discrete optimization. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP 99*, volume 1713 of *Lecture Notes in Computer Science*, pages 28–42. Springer, October 1999.
14. Toby Walsh. Search on high degree graphs. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 266–274. Morgan Kaufmann, August 2001.
15. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *12th International Conference on Distributed Computing Systems (ICDCS-92)*, pages 614–621, 1992.

16. Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 401–408. MIT Press, 1996.
17. Weixong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1–2):55–87, January 2005.