Stoch-DisPeL: Exploiting Randomisation in DisPeL

Muhammed Basharu, Inés Arana, and Hatem Ahriz

School of Computing
The Robert Gordon University, Aberdeen, UK
{mb,ia,ha}@comp.rgu.ac.uk

Abstract. We present Stoch-DisPeL, an extension of the distributed constraint programming algorithm DisPeL which incorporates randomisation into the algorithm. We justify the introduction of stochastic moves and analyse its performance on random DisCSPs and on Distributed SAT problems. We also empirically compare Stoch-DisPeL's performance to that of DisPel and DSA-B1N - our improved version of DSA. The results obtained show a clear advantage of the introduction of random moves in DisPeL. Our new algorithm, Stoch-DisPeL, also performs better than DSA-B1N.

1 Introduction

DisPeL [3, 2] is an algorithm which solves distributed constraint problems using a value-penalty mechanism in order to escape local optima. Since DisPeL is deterministic, it is vulnerable to the effects of poor random initialisations. Thus, given the exact same problem, DisPeL can find a solution with less than 100 iterations on some initialisations and requires about 10,000 iterations on others. In the worst case, DisPeL may not find a solution at all.

In this paper, we present Stoch-DisPeL, a modification to DisPeL that introduces randomness into its deadlock resolution strategy. We show that this randomisation improves performance while reducing both its memory requirements, and the complexity of its deadlock resolution process.

The remainder of this paper is structured as follows. First, we review some strategies for exploiting randomisation in combinatorial search, including DSA-B1N, our own extension to DSA [12, 13] in Section 2. Next, we briefly introduce DisPeL in section 3. Stoch-DisPeL is presented in Section 4. Finally, we present results of empirical experiments where we compare Stoch-DisPeL with DisPeL and DSA-B1N in Section 5.

2 Exploiting randomisation in combinatorial search

Hoos and Stutzle [7], analysed the behaviour of Stochastic Local Search (SLS) algorithms using Random Length Distribution (RLD) plots, including the effect of random initialisations and inbuilt random decisions.

They concluded that some initialisations led to solutions with short runs irrespective of algorithm parameter settings and that search efficiency actually decreases over time. They showed that once optimal cut-offs (i.e. number of iterations between restarts) were found, periodic resets dramatically improved the performance of the underlying algorithms, thus increasing the probability of finding solutions.

Related work by Hutter et al [9] on dynamic local search algorithms for solving SAT formulae introduces a scheme where, with a small probability, weights on constraints are smoothened towards the average weight. They showed that this randomisation can reduce the complexity of weight update procedures and still allow the underlying weighted hill-climber to outperform the more complicated algorithms.

In a similar study on complete backtracking algorithms [4], a new heuristic evaluation function for determining the next variable to label in a tie-break situation is presented. This heuristic increased the number of random decisions made at branching points which, combined with periodic restarts, boosted the performance of the underlying algorithms with speed-ups of several orders of magnitude.

In distributed constraint reasoning, a handful of similar randomisation strategies have been explored. DBA [11] is a distributed iterative improvement algorithm where, at each cycle, the agent which proposes the change leading to the largest improvement is allowed to modify its value. Wittenburg [10], proposed some non-deterministic tie-breaking schemes for this algorithm in which agents occasionally override their coordination heuristic thus enabling connected agents to change values simultaneously and, at the same time, making tie-breaking non-deterministic.

The Distributed Stochastic algorithm (DSA) [12, 13], is a distributed iterative improvement search algorithm, that relies on stochastic decisions to avoid deadlocks. In each iteration of DSA, each agent decides individually either to select a value that minimises the number of constraints it violates (with probability α) or to do nothing (with probability $1 - \alpha$) - where α is the probability of parallelism. In addition, deadlocked agents can make sideway moves that do not worsen their evaluations. Later work by Arshad and Silaghi [1] introduced DSA-B1, which incorporated additional randomisation by allowing agents to make uphill moves with a probability ($p2$). The algorithm was further improved resulting in the Distributed Simulated Annealing (DSAN) algorithm where $p2$ decays overtime [1].

We created DSA-B1N, a modified version of DSA-B1 which prevents agents from making uphill movements whenever their variables have consistent assignments. We found that DSA-B1N had improved performance when compared to DSA-B1 because the stabilisation of consistent variables allowed for increased search intensification activity. DSA-B1N also performed better than DSAN. Therefore, in the experiments reported in this paper we used DSA-B1N to represent the class of DSA algorithms. Furthermore, from our evaluations, we found that it solved the highest percentage of problems with $p2 = 0.05$ (with random DisCSPs) and

$p2 = 0.2$ (with distributed SAT problems[1]). We use these settings for the experiments with DSA-B1N reported in section 5.

3 DisPeL: a penalty-driven algorithm for solving DisCPSs

DisPeL [3, 2] is a successful distributed constraint satisfaction algorithm where each agent controls just one variable[2] and the objective is to find the first solution that satisfies all constraints. It is an iterative improvement algorithm, in which agents take turns to improve a random initialisation in a fixed order.

DisPeL (see Algorithms 1, 3, 2 and 4) uses penalties to modify underlying cost landscapes in order to deal with local deadlocks. These penalties are attached to individual domain values, and are used in a two phased strategy as follows:

1. In the first phase, the solution is perturbed with a *temporary penalty* in an attempt to force agents to try other combinations of values, and allow exploration of other areas of the search space(Algorithm 2 lines 5-8).
2. If the perturbation fails to resolve a deadlock, resolution moves to the second phase, where agents try to learn about and avoid the value combinations that caused the deadlock by increasing the *incremental penalties* attached to the culprit values (Algorithm 2 lines 9-11).
3. Whenever an agent detects a deadlock and has to use a penalty, it imposes the penalty on its current assignment and asks its neighbours to impose the same penalty on their current assignments as well (Algorithm 2 lines 8 and 11).
4. A no-good store is used to keep track of deadlocks encountered, and hence, used to help agents decide what phase of the resolution process is initiated a deadlock is encountered.

DisPeL is a synchronised iterative improvement algorithm in the sense that in each iteration, agents take turns to use the min-conflicts heuristic (or local repair) to select values that minimise the number of constraints violated. The order in which their turns are taken is decided using the Distributed Agent Ordering scheme [5]. Therefore, at initialisation, each agent locates its position in the ordering by locally partitioning its neighbours into parents (γ^+) and children (γ^-) using their lexicographic tags (or IDs). The cost function for each agent includes two types of penalties, as follows:

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

where:
d_i is the ith value in the variables domain

[1] Where the uphill move is simply a flip of the truth assignment.

[2] In this paper, we often use the term agent to also refer to the variable an agent represents.

Algorithm 1 DisPeL: Agent main loop

```
1: initialise
2: repeat
3:    messages ← accept()
4:    while active do
5:       penaltyRequest ← null
6:       processMessages()
7:       if cost function (h) is distorted then
8:          reset all incremental penalties
9:       end if
10:      if penaltyRequest ≠ null then
11:         respond_to_message()
12:         penaltyRequest ← null
13:      else
14:         if current value is consistent then
15:            reset all incremental penalties
16:            penaltyRequest ← null
17:         else
18:            check_for_deadlocks()
19:         end if
20:      end if
21:      send message(id, value, penaltyRequest) to neighbours
22:   end while
23: until termination condition met
```

Algorithm 2 procedure **check_for_deadlocks()**; initiating deadlock resolution.

```
1: if agentView(t) ≠ agentView(t-1) then
2:    select value minimising cost function
3:    return
4: end if
5: if agentView(t) is not in no-good store then
6:    impose temporary penalty on current value
7:    add agentView(t) to no-good store
8:    penaltyRequest ← ImposeTemporaryPenalty
9: else
10:   increase incremental penalty on current value
11:   penaltyRequest ← IncreaseIncPenalty
12: end if
13: select value minimising cost function
```

Algorithm 3 procedure **respond_to_message()** Responding to a penalty message received from a higher priority agent.

```
1: if penaltyRequest = ImposeTemporaryPenalty then
2:     increase incremental penalty on current value
3: else
4:     impose temporary penalty on current value
5: end if
6: select value minimising cost function
```

Algorithm 4 procedure **processMessages()**

```
1: for i = 0 to num(messages) do
2:     update AgentView with message.variable, message.value
3:     if message.penaltyRequest ≠ null then
4:         if message.penaltyRequest = IncreaseIncPenalty then
5:             penaltyRequest ← IncreaseIncPenalty
6:         else
7:             if penaltyRequest ≠ IncreaseIncPenalty then
8:                 penaltyRequest ← ImposeTemporaryPenalty
9:             end if
10:        end if
11:    end if
12: end for
```

$v(d_i)$ 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 ($t = 3$) [3].

4 Stochastic DisPeL

As we pointed our earlier, DisPeL's efficiency largely depends on the random initialisation used. Although DisPeL can benefit from a periodic restart strategy with new random instantiations, it is difficult to automatically determine appropriate cut-offs a priori. As an alternative, we try to exploit randomisation in DisPeL by focusing on the critical choice point in its deadlock resolution strategy, by making the choice of what phase to implement a random one.

4.1 The Stoch-DisPeL algorithm

In Stoch-DisPeL , we have changed all agents' behaviour so that whenever an agent is at a quasi-local-minimum, it decides randomly either to perturb the solution by imposing a temporary penalty (with probability

[3] This value for t is used in all our experiments irrespective of the problem size.

p) or to increase the incremental penaltiy (with probability $1-p$); rather than following the deterministic route of perturbing first and learning with incremental penalties later. This eliminates the need for the no-good store, since agents no longer have to determine if a deadlock was previously encountered, and thus reduce the algorithm's memory requirements and the number of operations agents have to implement when deadlocks are encountered.

We call this new algorithm Distributed Stochastic Penalty Driven Search (Stoch-DisPeL), and implement its new stochastic behaviours by replacing the **check_for_deadlock()** procedure listed in Algorithm 2 with the one outlined in Algorithm 5. All other processes executed by agents in DisPeL remain the same. Therefore, when an agent chooses the penalty to implement, it will still send a request to the affected neighbours to implement the same. Similarly, agents at the receiving end will still act in the same deterministic manner of prioritising the incremental penalty requests over temporary penalties.

Algorithm 5 Stoch-DisPeL: **procedure check_for_deadlocks()**

```
1: if agentView(t) ≠ agentView(t-1) then
2:     select value minimising objective function
3:     penaltyRequest ← null
4:     return
5: end if
6: r ← random value in [0..1]
7: if r < p then
8:     impose temporary penalty on current value
9:     penaltyRequest ← ImposeTemporaryPenalty
10: else
11:    increase incremental penalty on current value
12:    penaltyRequest ← IncreaseIncPenalty
13: end if
14: select value minimising objective function
```

4.2 Determining an optimal p value

The new parameter (p) in Stoch-DisPeL influences the behaviour and overall performance, by determining how often either of the penalties are used to resolve deadlocks.

A series of experiments, using RLD analysis, were run to evaluate the influence of the probability p on performance with distributed graph colouring problems and random DisCSPs. We specifically used RLD plots so that we could abstract out all influences of problem structure and random initialisations from performance; and therefore focus on the effects of different values for p between 0.1 and 0.9 (in steps of 0.1). In addition, a second experiment was also carried out where the most promising values from the first experiments were further evaluated on a larger dataset.

In the RLD plots in Figure 1, we ran Stoch-DisPeL on a single problem instance to compare the effect of the different values of p. In all cases we started the runs from the same random initialisation, so that the only influence on behaviour is the random choice made when deadlocks are encountered (i.e. p). In Figure 1, the plots[4] show the distribution of search costs on a single distributed graph colouring instance ($n = 100, k = 3, d = 4.6$) for the different values of p. For each value, 500 attempts were made with a maximum limit of 10,000 iterations before an attempt was deemed unsuccessful. The average and median costs from these runs are shown in Table 1.

p	average cost	median cost
0.1	292.0	197.0
0.2	282.3	206.5
0.3	259.1	203.0
0.4	288.1	209.5
0.5	302.4	228.5
0.6	322.2	245.0
0.7	367.3	290.5
0.8	395.0	309.0
0.9	545.5	451.0

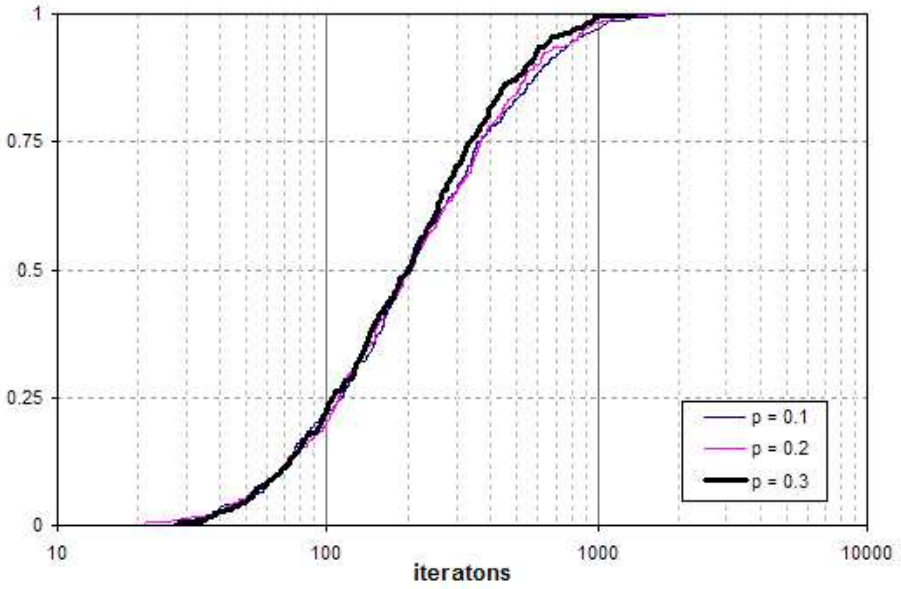
Table 1. Average and median search costs in Stoch-DisPeL from RLD analysis in Figure 1, for different values of p.

While the average costs in Table 1 vary, performance of the algorithm is almost identical for values of p from 0.1 to 0.5. and the median costs are also comparable. Pairwise Student t-tests for the values show that the distributions are mostly identical. And from $p = 0.6$ onwards search costs increase steadily.

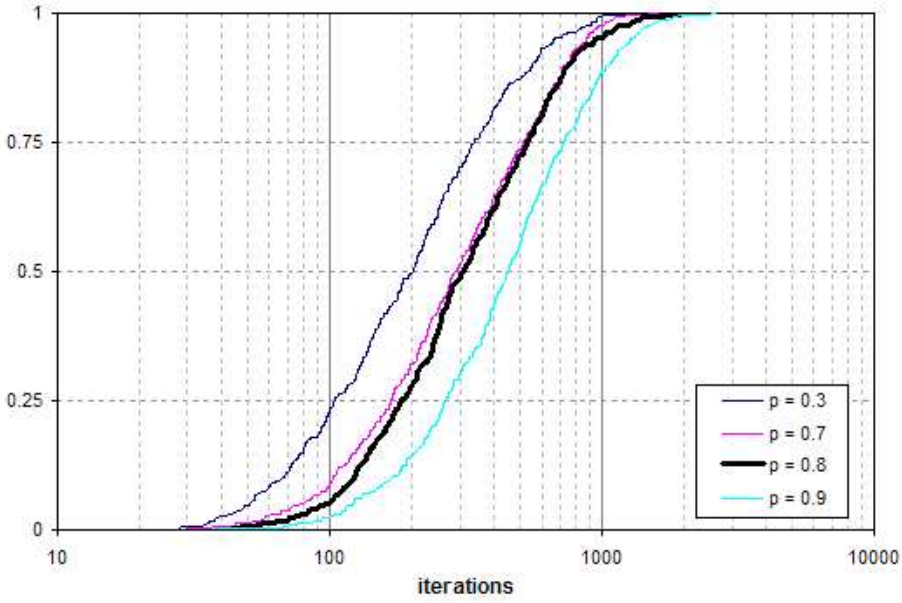
The same experiment was repeated using a random DisCSP instance ($< n = 60, d = 15, p1 = 0.1, p2 = 0.6 >$). The results, not shown here, follow a similar pattern with the earlier ones, i.e. any value from the range 0.1 to 0.4 appeared appropriate as they all produce similar (low) costs.

In order to confirm our hyphothesis for a low value for p, a second experiment was carried out where we tested the algorithm on a larger dataset with problems of different sizes, using small values for p. The results, showed no clear winner, but $p = 0.3$ appeared to be marginally better. Given that our results suggest that a value of p between 0.1 and 0.4 is appropriate, and a value of 0.3 gave marginally better results, we will use this value in our evaluation of Stoch-DisPeL.

[4] Several plots are used for the sake of clarity, as most curves overlap each other.



(a)



(b)

Fig. 1. Run Length Distribution of Stoch-DisPeL on a distributed graph colouring instance with different values for p.

4.3 Stoch-DisPeL vs. DisPeL

We believed that randomisation can allow the search get out of bad trajectories by simply changing the way penalties are selected. We confirmed this by running DisPeL several times on different problems to find bad instantiations. Then, ran Stoch-DisPeL on the same problems starting it off these instantiations. In Figure 2 an example of one such runs is shown, where we plot the Run Length Distribution (the "bad start" curve) of Stoch-DisPeL on a random DisCSP ($< n = 80, d = 15, p1 = 0.1, p2 = 0.5 >$) which DisPeL was unable to solve given the particular initialisation. In this case, Stoch-DisPeL was successful in each attempt within the allotted time of 8,000 iterations and did not in anyway suffer from the effects of the initialisation. The "good start" curve in the figure is a repeat of the same experiment, this time using an initialisation with which DisPeL found a solution after 41 iterations. Both curves are nearly identical and Stoch-DisPeL was unable to capitalise on the "good" initialisation. It is worth noting that while Stoch-DisPeL found more solutions, it sometimes needed more cycles to reach a solution.

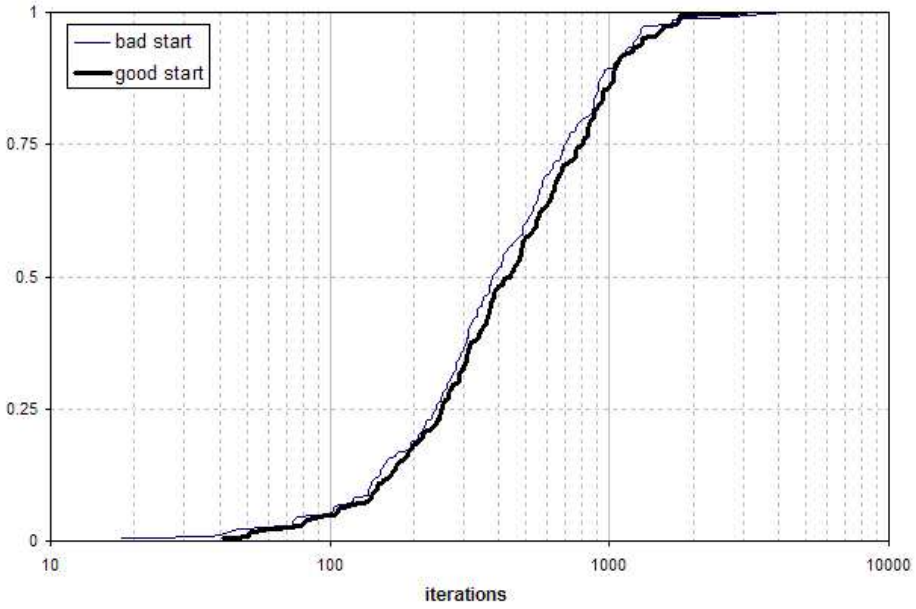


Fig. 2. Run Length Distribution of Stoch-DisPeL on a problem instance repeatedly starting with "good" and "bad" random initialisations.

5 Empirical Evaluation

In order to evaluate Stoch-DisPeL , we compared its performance with DisPeL's and DSA-B1N's on random DisCSPs and SAT formulae from the SATLib dataset [8][5]. In each case, we analysed the percentage of problems solved within the time limits and the costs (in terms of iterations) incurred in solving these problems.

5.1 Solving Random DisCSPs

Random binary DisCSPs of different sizes ($30 \leq n \leq 100$)were used in the evaluation of the algorithms to study how search costs scale up with the problem size. For each n, 100 instances were created where the ratio of constraints to variables was constant at 3:1 and the tightness ($p2$) of each constraint fixed at 0.5. There were 10 values in each variables' domain and all algorithms were limited to a maximum of $100n$ iterations on each attempt. The results of these experiments are plotted in Figures 3, 4, and 5, showing the success rates, median and average search costs respectively.

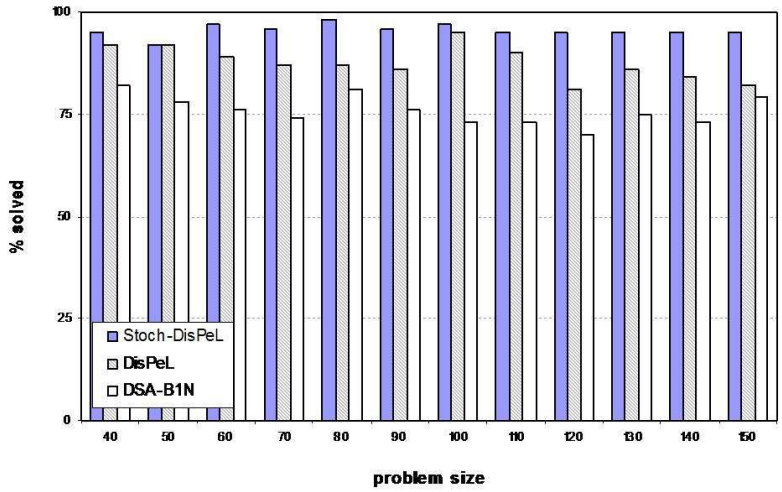


Fig. 3. Success rates on attempts on random DisCSPs of different sizes for Stoch-DisPeL, DisPeL and DSA-B1N

Figure 3 shows that Stoch-DisPeL is dominant over the other algorithms. It consistently solved more problems than both algorithms, except in the

[5] These problem instances are available online at http://www.satlib.org (accessed 23 January 2006).

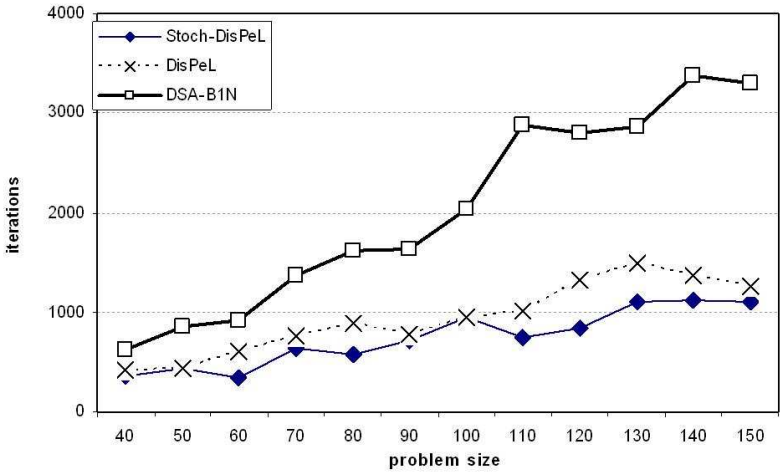


Fig. 4. Median search costs from runs in Figure 3.

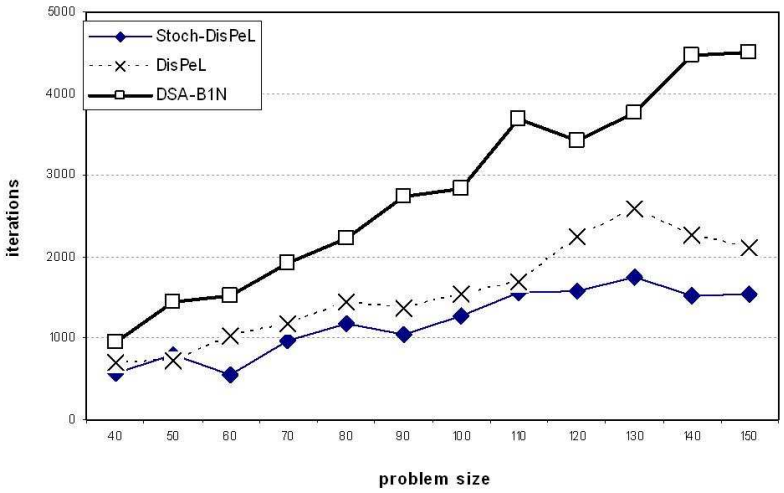


Fig. 5. Average search costs from runs in Figure 3.

$n = 50$ dataset where it was matched by DisPeL. Against DSA-B1N, both versions of DisPeL faired well. They consistently solved more problems than DSA-B1N and required fewer iterations especially on the larger problems (Figures 4 and 5).

5.2 Solving Distributed SAT problems

As we mentioned earlier, the algorithms were also evaluated with publicly available SAT instances. These were modelled as DisCSPs where each agent represents a literal (variable) and has to find a truth assignment that simultaneously satisfied all relevant clauses given the current assignments of other agents appearing in those clauses. SAT was chosen because, amongst other things, it is a domain where stochastic algorithms have traditionally done well, especially in centralised local search.
Datasets with 75, 125, 150, 175 literals per problem where used in the experiments, each with 100 solvable instances. And we also used the first 500 instances from the dataset with 100 literal problems. In all cases, the algorithms were limited to a maximum of $100n$ iterations (where n is the number of literals in a formula) before attempts were deemed unsuccessful. The results of these experiments are summarised in Figures 6, 7 and 8.

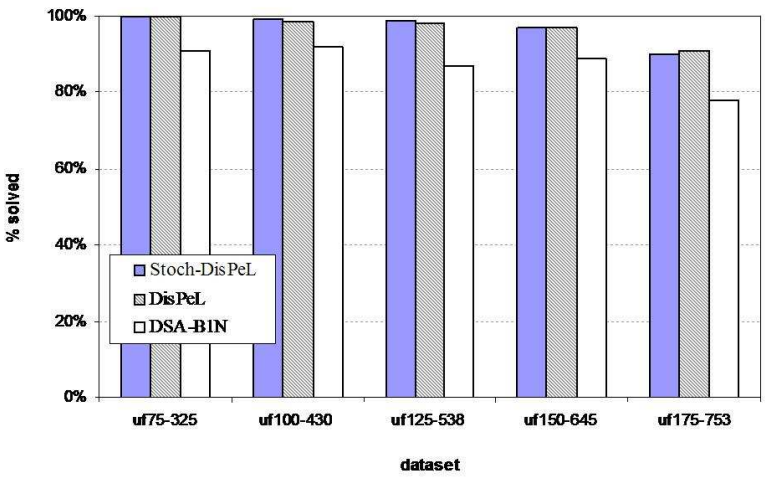


Fig. 6. Percentage of problems solved by Stoch-DisPeL, DisPeL, and DSA-B1N from attempts on benchmark SAT instances.

The plots in the figures show that, in terms of success rates, Stoch-DisPeL and DisPeL are evenly matched. But Stoch-DisPeL has a slight cost advantage over DisPeL and, on this criterion, the results are fairly

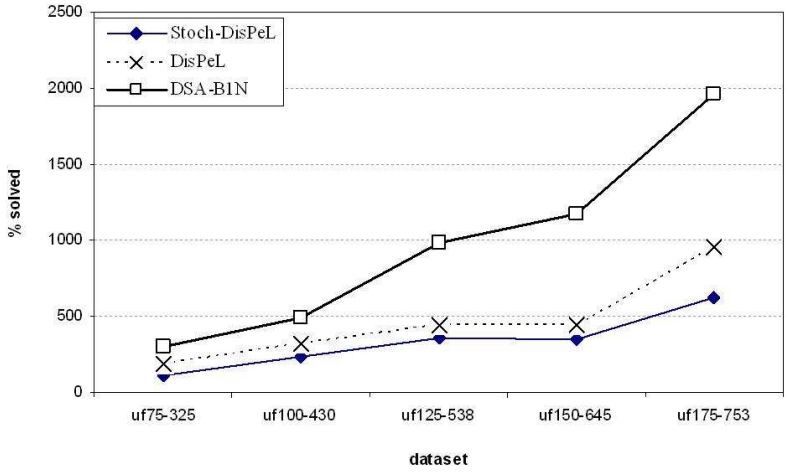


Fig. 7. Median costs (iterations) of Stoch-DisPeL, DisPeL, and DSA-B1N used-up to solve the problems in Figure 6.

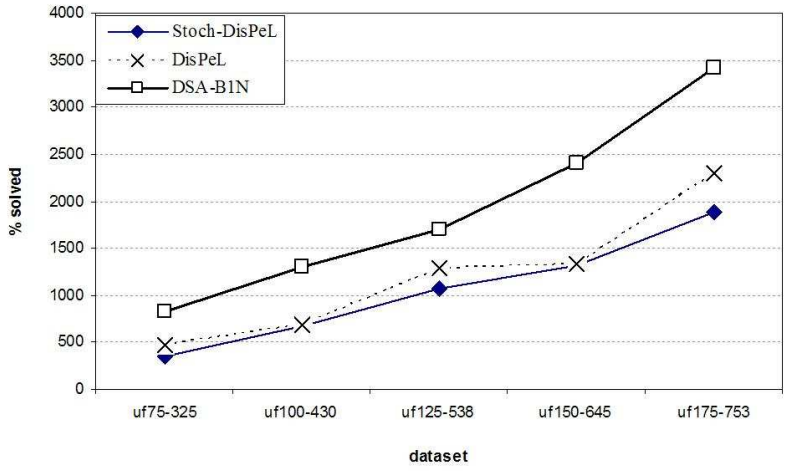


Fig. 8. Average costs (iterations) of Stoch-DisPeL, DisPeL, and DSA-B1N used-up to solve the problems in Figure 6.

consistent with those from the experiments on random DisCSPs. Both algorithms however do well compared to DSA-B1N especially as the problems get larger. In the experiments with the original DSA algorithm on the same datasets, Hirayama and Yokoo [6] reported results showing a dismal performance in the domain. At best the algorithm was only able to solve 11% of the problems in one dataset. The stronger results reported here confirm the necessity and the profound impact of the occasional non-improving moves that are used to help the algorithm deal with local optima.

6 Conclusions

We have described Stoch-DisPeL, a stochastic variation of DisPeL which introduces a random choice in DisPeL's deadlock resolution strategy. Rather than follow the fixed rule of perturbing and then incrementing penalties, in Stoch-DisPeL agents randomly decide on which type of penalty to use. Therefore, agents decide to use the temporary penalty with a probability p and the incremental penalty with $1 - p$. As a result of this modification, the no-good store is no longer used thereby reducing the memory requirements and the number of operations performed by agents when deadlocks are discovered.
We showed, from empirical tests, that the performance of the new algorithm was optimal at small values of its critical parameter ($0.1 \leq p \leq 0.4$). The algorithm was evaluated on random DisCSPs and benchmark instances of the boolean satisfiability problem. Performance was viewed against results from DisPeL and an improved version of the Distributed Stochastic Algorithm, DSA-B1N. The results showed that randomisation boosted the performance of the penalty based strategy. Stoch-DisPeL consistently solved more problems than DisPeL and DSA-B1N, and it typically required fewer iterations in the process.
Stoch-DisPeL solved more problems than DisPeL, but in some cases it incurred higher costs. Against this background, a hybrid of both algorithms, exploiting the best features of each, is an attractive proposition. Probably done in a way that allows agents use the deterministic approach early in the search and increasingly choose the stochastic approach as the process draws on.

References

1. Muhammad Arshad and Marius C. Silaghi. Distributed simulated annealing and comparison to DSA. In *Proceedings of the Fourth Workshop on Distributed Constraint Reasoning, IJCAI-DCR 2003*, 2003.
2. Muhammed Basharu, Inés Arana, and Hatem Ahriz. Escaping local optima with penalties in distributed iterative improvement search. In Amnon Meisel, editor, *Proceedings of DCR 05 - the Sixth Intenational Workshop on Distributed Constraint Reasoning*, pages 192–206, 2005.

3. Muhammed Basharu, Inés Arana, and Hatem Ahriz. Solving DisC-SPs with penalty-driven search. In *Proceedings of AAAI 2005 - the Twentieth National Conference of Artificial Intelligence*, pages 47 – 52. AAAI, 2005.
4. Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, July 1998. AAAI Press / The MIT Press.
5. Youssef Hamadi, Christian Bessière, and Joël Quinqueton. Backtracking in distributed constraint networks. In Henri Prade, editor, *13th European Conference on Artificial Intelligence ECAI 98*, pages 219–223, Chichester, August 1998. John Wiley and Sons.
6. Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. *Artificial Intelligence*, 161(1–2):89–115, January 2005.
7. Holger H. Hoos and Thomas Stützle. Evaluating las vegas algorithms: Pitfalls and remedies. In Gregory F. Cooper and Serafín Moral, editors, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI'98)*, pages 238–245. Morgan Kaufmann, July 1998.
8. Holger H. Hoos and Thomas Stutzle. Satlib: An online resource for research on SAT. In I. P. Gent, H. van Maaren, and T. Walsh, editors, *Third Workshop on the Satisfiability Problem (SAT 2000)*, pages 283–292. IOS Press, 2000.
9. Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In P. Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP02)*, volume 2470 of *LNCS*, pages 233–248, London, UK, September 2002. Springer-Verlag.
10. Lars Wittenburg. Distributed constraint solving and optimizing for micro-electro-mechanical systems. Master's thesis, Technical University of Berlin, December 2002.
11. 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.
12. Weixiong Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization:parallelism, phase transitions and performance. In *Proc. AAAI Workshop on Probabilistic Approaches in Search*, pages 53–59, 2002.
13. 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.