Modifying landscapes with penalties in iterative improvement for solving distributed constraint satisfaction problems.

BASHARU, M.B.

2006

MODIFYING LANDSCAPES WITH PENALTIES IN ITERATIVE IMPROVEMENT FOR SOLVING DISTRIBUTED CONSTRAINT SATISFACTION PROBLEMS

MUHAMMED BASHIRU BASHARU

A thesis submitted in partial fulfilment of the
requirements of
The Robert Gordon University
for the degree of Doctor of Philosophy

Supervised by Dr. Ines Arana and Dr. Hatem Ahriz

September, 2006

BEST COPY AVAILABLE

VARIABLE PRINT QUALITY

Abstract

The recent growth of the internet and distributed computing facilitated by the internet has created more opportunities for collaboration between individuals and organisations. These new forms of collaborative activity put groups of participants in situations where there is a shared objective but at the same time there is also a competition for resources by the participants. Hence, there is a need for participants to make compromises in order to reach agreement. Examples of such situations include collaborative scheduling in supply chain management or even individuals trying to agree on a schedule of meetings. The process of reaching agreement on any of such situations can be automated and the first step in such automation may be to model the situations as Distributed Constraint Satisfaction Problems (DisCSPs).

DisCSPs formally describe distributed problems where each participant in the problem is represented by an agent, and the collection of agents have to collaborate in order to reach a satisfactory agreement (or find a solution) for a problem. Yokoo's seminal work on distributed constraint solving introduced the idea of distributed approaches to solving such problems. And following that, research in the new field has come up with a variety of techniques, including combinatorial search and other forms of inference, for solving DisCSPs. In this study, we investigate an iterative improvement search approach for solving DisCSPS.

Iterative improvement search has the advantage, over constructive search, of being able to converge quicker on large problems. But, it also has a propensity to converge to local optima in the process. Previous work on iterative improvement search (in centralised and distributed forms) has considered a variety of techniques for dealing with the problem of convergence on local optima. Prominent amongst these include introducing forms of randomisation or modifying the shapes of objective landscapes to guide a search out of plateaus. We present a new approach for dealing with local optima in distributed iterative improvement by modifying the cost landscape with two forms of penalties which are attached to individual domain values. We use one type of penalty to perturb solutions and the other to help agents learn about (and avoid) domain values frequently associated with local optima. We argue that, compared to other forms of landscape modification, our approach has a more dramatic effect on cost landscapes and hence, it is a more effective strategy for solving DisCSPs by iterative improvement search.

Based on the idea of using two forms of penalties for dealing with local optima, we introduce three new distributed algorithms for solving DisCSPs where the objective is to find the first solution that satisfies all constraints simultaneously. First, we introduce the Distributed Penalty Driven Search (DisPeL) which is built around a two phased strategy of using penalties. We also introduced a stochastic variation of that algorithm (Stoch-DisPeL), which reduces some of DisPeL's complexities and allows agents to randomly

decide which of the two penalties to use when dealing with local optima. These two algorithms are specifically designed for scenarios where each agent represents a variable in a DisCSP, unlike our third algorithm (Multi-DisPeL) which extends some of the ideas from the earlier algorithms to DisCSPs where each agent represents multiple variables.

We evaluated all three algorithms on a number of distributed constraint satisfaction problems including distributed graph colouring, distributed Boolean satisfiability, and random DisCSPs. We also compared them to state-of-the-art distributed iterative improvement algorithms. The results of the evaluations show that the penalty driven algorithms are effective alternatives; they solved more problems and typically incurred lower costs in the process.

Acknowledgements

I am deeply indebted to my supervisors Dr. Ines Arana and Dr. Hatem Ahriz for their guidance and support during my PhD study. I was very fortunate to have them both as supervisors. I thank them for giving me the opportunity to work with them, for their patience, and for all the interesting discussions we had over the three years of my research. Without them, this thesis would be unreadable. I also thank my examiners, Dr. Christian Bessiere and Dr. Roger McDermott, for their time and advice.

The staff at the School of Computing have been marvellous during my time there. I would like to thank Colin and Caroline for always allowing me use all the processors in my vicinity to perform my experiments. Thank you to Kathy, Ann, Diane, and Marie for their constant and cheerful assistance. I especially want to thank Prof. Susan Craw for her support.

I have to say thank you to all my colleagues at the CTC/Smartweb, who all made the time at Aberdeen an enjoyable and memorable one. I especially like to thank Zia, Murat, Siddartha, Dietrich, Ratiba, Fiona, Ralf, Bhavani, Daniel, Stewart, Ivan, Nirmalie, Rahman, Stella, Selpi, Bayo, Sunny, and Kefang for their friendship.

The most important thank yous are reserved for my parents (i.e. mum and dad), who have always supported and encouraged me to pursue my whims and aspirations. I don't think I can ever thank them enough. I also want to thank my sister, Halima, who always had a nice motivating comment to cheer me up during the write up. And, to my brothers, Ibrahim and Yusuf, for their continuing support and encouragement.

This study was fully funded by a research studentship from the School of Computing, The Robert Gordon University, Aberdeen.

Declarations

This thesis has been compiled by myself and describes my own work. All mention of other work have been duly cited in the bibliography.

Some parts of the work presented in this thesis have appeared in the following publications:

Chapter 5

Muhammed Basharu, Ines Arana, and Hatem Ahriz. Escaping local optima with penalties in distributed iterative improvement search. In Amnon Meisel, editor, *Proceedings of DCR 05 - the Sixth International Workshop on Distributed Constraint Reasoning*, pages 192-206, 2005.

Muhammed Basharu, Ines Arana, and Hatem Ahriz. Solving DisCSPs with penalty-driven search. In *Proceedings of AAAI 2005 - the Twentieth National Conference of Artificial Intelligence*, pages 47-52. AAAI, 2005.

Muhammed Basharu, Ines Arana, and Hatem Ahriz. Distributed Guided Local Search for Solving Binary DisCSPs. In Ingrid Russell and Zdravko Markov, editors, *Proceedings of FLAIRS 2005 - the Eighteenth International Florida Artificial Intelligence Research Society Conference*, pages 660-665, 2005.

Chapter 6

Muhammed Basharu, Ines Arana, and Hatem Ahriz. Stoch-DisPeL: Exploiting randomisation in DisPeL. In *Proceedings of DCR 06 - the Seventh International Workshop on Distributed Constraint Reasoning*, 2006.

Muhammed Basharu

Contents

List of Figures

List of Tables

List of Abbreviations

AWCS	Asynchronous Weak Commitment Search
CSP, CSPs	Constraint Satisfaction Problem(s)
DBA	Distributed Breakout Algorithm
DisBO	Distributed Breakout
DisBO-wd	Distributed Breakout with Weight Decay
DisCSP, DisCSPs	Distributed Constraint Satisfaction Problem(s)
DisPeL	Distributed Penalty Driven Search
DSA	Distributed Stochastic Algorithm
Multi-AWCS	Asynchronous Weak Commitment Search for Agents with Multiple Local Variables
Multi-DB	Distributed Breakout Algorithm for Agents with Multiple Local Variables
Multi-DisPeL	Distributed Penalty Driven Search for Agents with Multiple Local Variables
RLD	Run Length Distribution
SAT	Boolean Satisfiability Formulae
Stoch-DisPeL	Stochastic Distributed Penalty Driven Search

Chapter 1

Introduction

Recent commentaries in the popular press suggest that we are currently in the midst of the Information Revolution. Just like the printing press, the steam engine, and television, the spread of the Internet has changed, and it is still expected to change, our lives profoundly and in unanticipated ways. More than any time in human history, nearly half of the world's population has instant access to information on just about any topic under the sun. Electronic mail and other Internet enabled technologies allow us to communicate instantly too. Suddenly, the degrees of separation between any two individuals on the planet have been shortened.

This instant connectivity between individuals is also changing the ways we collaborate and work together. For example, open source projects like the Linux operating system or the Wikipedia project have shown us how geographically dispersed individuals can successfully collaborate on ad hoc basis. And, nowhere is this impact more felt than in the business environment. The ways companies operate, compete, and collaborate are constantly being changed. Value chain networks that interconnect internal information systems via the Internet now provide an end-to-end link between end-users and raw material suppliers such that companies can respond quicker to changes in the market place. In fact, it has come to the point that explicit virtual organisations can now exist; where companies at different levels of a value chain can rapidly form new alliances to exploit specific opportunities for very short periods of time. Unlike hierarchical alliances in existing

supply chain networks, such ad hoc alliances face some quite complex challenges; from decentralised coordination to trust - since partners in one alliance may become competitors months after the dissolution of the alliance.

In its broadest sense, this study is about such collaborations. It is interested in the forms of collaborations where there is a common objective to be achieved by a group of participants and yet each participant has its own objectives. Furthermore, in these collaborative situations there is a competition for resources and trust between participants is not unbounded. Such collaborations permeate different levels of human endeavour, from individuals trying to schedule meetings to joint multinational projects like the International Space Station. In particular, this study is interested in how problems are solved in such collaborative groupings. We consider problems like scheduling or resource allocation where group objectives are clearly defined but individual objectives introduce additional limitations on how solutions are negotiated. We consider how such problems can be solved automatically by systems of autonomous and homogeneous software agents representing participants in a group.

Normally, in the absence of individual objectives, such problems already belong to a class of computationally intractable problems where the difficulty of a problem can increase exponentially with its size. At present, there are no known algorithms that guarantee that such problems can be solved in reasonable time. However, over the last thirty years Constraint Satisfaction has emerged as a successful paradigm for dealing with these problems. The constraint satisfaction approach uses the constraints implicit in a problem to rule out parts of a problem that can not be in a solution and as such improve the efficiency of the problem solving process. As such, in constraint satisfaction, a problem is first formally represented as a Constraint Satisfaction Problem (CSP) comprising a finite set of decision variables, each with a set of alternatives it can adopt, and a set of constraints [23]. The constraints in a problem define relations between variables, and state which alternatives the decision variables can simultaneously assume. In scheduling for example, a typical constraint is one that imposes precedence relationships on the order in which any set of tasks are to be performed. A CSP is solved when all choices for decision

variables are consistent with all the constraints between the variables.

In collaborative situations, the problems to be solved can be formally modelled as Distributed Constraint Satisfaction Problems (DisCSPs) [122], where, in addition to the CSP components, there is a set of autonomous agents. Each agent represents an individual participant in the grouping, controlling all decision variables owned by that participant. In the DisCSP model, it is assumed that participants are physically dispersed and, for several reasons, information about a problem remains in the hands of its owners and as such all information can not be collected at a single location. Agents collaborate to solve a DisCSP by negotiating on possible choices for their decision variables, based on information available locally, to find a stable state where all choices are consistent with all the constraints between variables.

Besides the intractability of DisCSPs, the distribution of information creates additional challenges for the process of solving them. For instance, agents can not appreciate the impact of their decisions on the grouping since each agent has partial knowledge of the problem being solved. As such, all decisions are based on the information held within each agent and the bits of information gathered from local connections with other agents. There is also a privacy requirement that limits the amount of information each agent is permitted to reveal to other agents. Therefore, the key research challenge in solving such problems remains crafting out behaviours and interactions for individual agents so that the problem solving efficiency of a system of agents is improved.

1.1 Research objective

DisCSPS are mostly solved by search. The techniques for carrying out such search fall under two major categories: (1) backtracking and (2) iterative improvement. Backtracking search constructs solutions by sequentially considering possible alternatives and revising early decisions that rule out all alternatives for later decisions. Backtracking is guaranteed to be complete as it can determine when a problem is unsolvable with certainty and, when it is solvable, find all possible solutions. Iterative improvement, on the other hand, starts its search at a random position in the space of all possible combinations of alternatives,

and it proceeds to move from one point to the next in that space making improvements until a solution is found.

In centralised problems, iterative improvement algorithms offer the key advantage of converging quicker on large problems than complete backtracking search; although without similar theoretical guarantees of completeness. Such algorithms have been shown to be remarkably effective for solving problems with millions of variables. For example, a local search algorithm has been used to solve the n-queens problem with a million queens in less than a minute [76], while the practical limit for complete backtracking is a few hundred queens [23].

The weaknesses of complete algorithms are amplified further when solving problems in distributed environments with the privacy restrictions of DisCSPs. The added cost of communications between processes imposes even more severe practical limits on such algorithms. Such is the effect of these limitations that in the literature on distributed backtracking most experiments are performed with problems having no more than 60 variables[1], while distributed versions of iterative improvement algorithms have shown promise on larger problems.

However, iterative improvement in general suffers from the problem of regular convergence to local optima i.e. non-optimal solutions or traps in the solution space that prevent a search from making any improvements and halts its progress. The strategy for dealing with local optima is thus a crucial component of any iterative improvement algorithm. Such strategies naturally aim to help a search find the quickest paths out of locally optimal regions and possibly try to prevent the search from returning to such regions. Therefore these strategies also influence the overall effectiveness and efficiency of the underlying iterative improvement search.

In this study, we focus on distributed iterative improvement search for solving DisCSPs. Our primary research objective is to improve performance of this form of search by improving the strategy for dealing with local optima. To do this, we look at cost landscape modification as a means of enabling a search to find paths out of such optima. We propose

[1]This was an observation from a personal survey of published literature in the field up to 2005.

a new strategy based on the idea of using penalties attached to individual domain values and argue that it is a more effective means of landscape modification. This strategy is used as the basis for three new distributed iterative improvement algorithms.

1.2 Key contributions

The key contributions of this study are:

1. A new mechanism for modifying cost landscapes with penalties on individual domain values. We argue that modifying cost landscapes with weights on constraints, which is a popular approach, may not be effective at inducing exploration in landscapes dominated by plateaus. Therefore, we propose a much finer grained approach where assignments associated with plateaus are penalised. This is extended further so that penalties can also be used to perturb searches as another means of encouraging search exploration.

2. The Distributed Penalty Driven Search (DisPeL) algorithm is introduced built around the new landscape modification mechanism. DisPeL is a synchronous distributed iterative improvement algorithm for solving DisCSPs where each agent has only one variable. In DisPeL, sequential improvements are made to a random initialisation and a two phased penalty strategy is used by agents to deal with deadlocks occurring at local optima. Penalties are used to perturb the solution in the first phase and are used to modify cost landscapes in the second phase. DisPeL is sound and it has a linear space complexity. Extensive experimentation has been carried out using different problems, with sizes of up to 200 variables, and the results show DisPeL provides significant cost savings over the Distributed Breakout algorithm [123]. We also show that it is robust to communications failures, where it still solves a high percentage of problems when up to 40% of messages are not received by intended recipients.

3. DisPeL is extended in form of the Stochastic Distributed Penalty Driven Search (Stoch-DisPeL) algorithm, in order to reduce the risks of bad initialisations in Dis-

PeL. Actions to resolve deadlocks in DisPeL are performed in a deterministic manner; and as such, if a search starts off on a trajectory that does not lead to a solution, DisPeL is unable to avoid an infinite oscillation between non-solution states. Stoch-DisPeL makes the choice of what resolution phase to implement a random one and therefore the search trajectory is no longer exclusively determined by the random initialisation.

4. Stoch-DisPeL is extended to create Penalty Driven Search for Agents with Multiple Local Variables (Multi-DisPeL). As its name implies, Multi-DisPeL is designed for problems where each agent has multiple variables, constraints between these variables, as well as constraints with variables owned by other agents. The penalty mechanism is integrated into the local algorithms used by agents and applied in order to resolve deadlocks that occur both between local variables and between variables belonging to different agents.

1.3 Scope of this study

Attention in this study is focused on algorithms for solving DisCSPs where the objective is to find the first solution that satisfies all constraints simultaneously. As such, we do not consider Distributed Constraint Optimisation Problems, where the objective is to find the best solution as determined by a stated objective function. Neither do we consider distributed problems that require an algorithm to return all possible solutions to it. Finally, we assume that each problem's specification is fixed in advance and does not change during the process of attempting to solve it.

1.4 Thesis outline

This thesis is organised as follows. Chapter 2 is a survey of related work on algorithms for solving Constraint Satisfaction Problems both in centralised and distributed environments. In Chapter 3, a formal description of the Distributed Constraint Satisfaction Problem is presented, as well the model of DisCSP used in this study. We also outline our assumptions

and the scope of the model. New ideas for modifying cost landscapes are presented in Chapter 4 along with a comparison with the dominant approach. In Chapters 5, 6, and 7 new distributed iterative improvement algorithms, based on the ideas in Chapter 4, are presented. Results of empirical evaluations are also presented in the respective chapters. Finally, a summary of thesis and suggestions for further work are presented in Chapter 8.

Chapter 2

Survey of Related Work

2.1 Constraint Satisfaction

Much of the work in Artificial Intelligence (AI) problem solving can directly or indirectly be placed into one of two categories: representation or search. In areas like machine vision, natural language, and expert systems, the challenge for AI is to find appropriate representations of the real world in forms that computers can manipulate meaningfully. While in other areas, such as machine learning, game playing, and planning, the AI challenge is to find patterns or valid series of actions from extremely large sets of possibilities. Constraint satisfaction (or constraint reasoning) is an emerging paradigm that brings both these two categories together, to deal with a wide variety of problems that require some intelligence to solve. In problems such as scene recognition [118], option trading [64], or resource allocation [58, 1, 39], the constraint satisfaction paradigm ties the two AI categories together by providing formal methods for describing problems as Constraint Satisfaction Problems (CSPs); a well as providing a host of techniques for solving such problems.

Formally, a CSP is defined as a triple (X, D, C) [23], where X is a set of finite decision variables, D is a set of finite domains listing possible values for each variable, and C is the set of constraints that restrict what values can be assigned to sets of variables. The solution to a CSP is an assignment of a value to each variable, so that all constraints in the CSP are simultaneously satisfied. CSPs are NP Complete problems that involve

combinatorial search spaces and are therefore solved by search and/or inference methods. In the following, we outline the main categories of algorithms for solving CSPs, highlighting the strengths and any limitations of algorithms in each category along the way.

2.1.1 Search techniques for solving CSPs

Search algorithms for solving CSPs may be classified as either constructive search or local search. A review of prominent work in these two categories of search algorithms is presented in this section. Given the direct relevance to this study, we dwell a bit on landscape modification techniques in the review of literature on local search. We also include a survey of cooperative search which is another form of distributed problem solving, but it differs from DisCSP solving in that cooperative approaches focus on integrating parallel processes to improve efficiency.

Systematic backtracking search

Backtracking search, which is one of the earliest methods used for solving CSPs, is generally described as an incremental process in which a partial solution is extended until a full solution is found [24]. The partial solution is a list of labelled variables (i.e. variables assigned values), that starts off with an empty list. This solution is extended by assigning a value to an unlabelled variable that satisfies all its constraints with variables in the partial solution. If a partial solution can not be extended (or a dead-end is reached), the search tracks backwards to revise earlier decisions to consider other alternatives for labelled variables. Backtracking search terminates when a solution has been found or it has determined that all possible combination of values in the smallest partial solution can not be extended to find a complete solution.

The basic backtracking algorithms (chronological backtracking) extends partial solutions in predefined orders; and whenever dead-ends are encountered, the search moves backwards one step to revise the last variable labelled. But, in doing so, backtracking can suffer from trashing i.e. where the search continues to revisit the same dead-end without revising the earliest decisions responsible for the dead-ends. Therefore the search wastes a

lot of effort to discover this and for this reason chronological backtracking is rarely used. In its place a number of modifications have been introduced which change, amongst other things, how far to backtrack when dead-ends are encountered and the order in which variables are labelled.

Trashing in backtracking can be dealt with using schemes that control how far the search backtracks when dead-ends are encountered or with schemes for learning about the causes of the dead-ends. Backjumping schemes [32, 22, 89] allow a search to jump over recently labelled variables, unconnected with a dead-end, to the most recent variable contributing to a domain wipe out i.e. all values in the domain of an unlabelled variable are ruled out by assignments to variables in the partial solution. On the other hand, learning schemes such as Ginsberg's Dynamic Backtracking [33] allow a search to identify the earliest causes of dead-ends and rectify them without unlabelling variables unrelated to the dead-end. A detailed review of backtracking algorithms can be found in [24].

The introduction of Forward Checking [46] schemes allow backtracking algorithms to anticipate the effect of early decisions on the search and propagate these decisions to unlabelled variables. When a variable is labelled, all values inconsistent with its assignment are removed from the domains of unlabelled variables. This helps the search identify potential deadlocks in advance when domains of unlabelled variables become empty. With Forward Checking there is also an opportunity to dynamically change the order in which variables are labelled or values are selected. For example, with the popular Smallest Domain First heuristic [46], the solution is extended with the unlabelled variable with the smallest remaining domain so that the search can focus on where it is likely to fail first.

Backtracking has the advantage of being complete. Its systematic exploration of the search space will guarantee that it correctly determines that either a problem has one or more solutions, or that there are no solutions to it. But, because of its exhaustive exploration of a space that is exponential to a problem's size, backtracking can be expensive and the time required to solve a problem may grow exponentially as well.

Local search techniques

Local search (or iterative improvement search) describes a form of search that starts off at a random point in a search (or state) space with a complete assignment of random values to all variables (i.e. a candidate solution); and in successive iterations, it explores different points in the space until a valid solution to a problem is found or the maximum time allowed has elapsed. It is termed local, in the sense that the search moves to adjacent points in the search space i.e. moving to candidate solutions reachable by changing the value of just one variable in the current candidate solution, as it progresses.

The basic local search algorithm is a greedy approach that moves, in each iteration, to the most improved the solution in the neighbourhood[1] of the current candidate solution, as determined by a given objective/cost function. For example, in CSP solving, a typical objective function is the number of satisfied constraints and therefore a valid solution (and the maximum objective) is one in which all constraints are satisfied. Alternatively, a cost function may be used to drive the search where the cost of a candidate solution is the number of constraints violated and a valid solution is one with zero cost.

A visual metaphor of rugged landscapes is typically used to describe this form of search where the space of possible solutions form an uneven landscape and the altitude of each point in this landscape (i.e. a candidate solution) is determined by its objective value or cost. The metaphor is extended further to describe the basic greedy algorithm as either hill-climbing or steepest descent search. Therefore, the greedy search is often described as improving uphill in the objective landscape (in the case of hill-climbing) or downhill in the cost landscape (for steepest descent) [2].

A hill-climbing search continues until either a solution is found or a local optimum is reached. A local optimum is a deadlock (or conflict) state in which some constraints are violated but the solution can not be improved by changing the value of any single variable i.e. there is no improvement in the neighbourhood of the current candidate solution. Visually, local optima are described as points on plateaus in objective/cost landscapes, in

[1] The neighbourhood of a solution is the set of all adjacent solutions reachable from the current candidate solution

[2] From Chapter 4 onwards, we assume that search is steepest descent in the cost landscape, hence any reference to improving moves imply downhill moves.

which the landscape is flat in all directions from the points.

On encountering a local optimum, the basic hill-climbing algorithm discards the current candidate solution and the search is restarted from a new location in the search space. Generally, hill-climbing is considered to be inefficient because of its strategy for dealing with local optima. Some of the early modifications to hill-climbing, particularly Simulated Annealing (SA) [60] and Tabu Search (TS)[34, 35], introduce more subtle mechanisms for avoiding and escaping from local optima so that the search experience gained is not entirely lost.

Drawing on an analogy with the annealing of metals, Simulated Annealing modifies the standard hill-climbing algorithm to accept some non-improving moves with a small probability. These random moves give a search opportunities to leave plateaus and also promote search space exploration so that regions not necessarily covered by a greedy hill-climber are considered. In addition, the non-improving moves can help a search avoid local optima in the first place by occasionally knocking it off search paths bound for such plateaus. The probability of accepting non-improving moves is a function of the annealing temperature and this temperature decays over time according to a cooling schedule. The main advantage of SA is that it has been proven to converge on global optima, albeit with infinite time [12]. Tabu-search, on the other hand, introduces a form of learning to hill-climbing and like SA, accepts the occasional non-improving move. In addition, TS tries to overcome the possibility of oscillation i.e. the same set of moves maybe repeatedly accepted, by maintaining a tabu list of recent moves. Moves listed as tabu may not be repeated for the duration of their stay on the list. SA and TS are amongst the oldest and the most widely used heuristics for solving combinatorial problems [12]. Other variations on the standard hill-climbing consider other forms of randomisation, in addition to new heuristics to guide the search. Examples include WalkSAT [102], GSAT [103], GRASP [63], Variable Neighbourhood Search [45], and Iterated Local Search [86].

Dealing with local optima by randomisation has the advantage of giving the search opportunities to make jumps to distant regions of the search space and expanding its scope. But, except in the case of tabu search, randomisation strategies are generally

memory less and are unable to prevent a search from repeating the same mistakes. Nor, do they focus adequately on the causes of deadlocks. Landscape modification schemes, which are popular in local search algorithms for solving boolean satisfiability formulae (SAT), try to alleviate this last problem by introducing naive learning mechanisms that aim to allow a search to remember plateaus encountered and to avoid other regions of the search space where solutions do not exist [13].

In the earliest work along this line, Morris [81] extends local search by introducing weights which are attached to constraints and are incorporated into a problem's objective function in the Breakout algorithm; such that the local search proceeds to minimise the sum of weighted violations. When the search is stuck at a local minimum, weights on violated constraints are increased - highlighting these constraints and changing the shape of the objective/cost landscape. Therefore, the search emphasis is on satisfying constraints with the highest weights, which are considered to be the most difficult to resolve. But, Morris admits that the constraint weights can modify the landscape to the extent that paths to solutions are blocked off, therefore, can leave the search to wander aimlessly in unprofitable regions. Later works on SAT solving consider other formulations that allow weights to decay over time. In [29, 30] weights are modified after each move by the search, so that those on violated constraints are increased and, at the same time, decay to enable the search forget previous weight increases and to focus on the most recent increases. Similarly, in the Scaling and Probabilistic Smoothing (SAPS) [57], a smoothing procedure is introduced that brings all weights towards the mean weight with a certain probability. Other algorithms for SAT solving like the Discrete Lagrange Multiplier (DLM) [116], the Smoothed Descent and Flood (SDF) [99], and Exponentiated Sub-Gradient (ESG) [100], which also use similar but more complicated weighting schemes have been shown to be very efficient.

It is argued that constraint weighting allows a search to learn or prioritise "important" constraints, but Tompkins and Hoos in [111] disagree with this and they argue that weights only serve as an effective diversification mechanism. They also add that weights do not hold important cues about difficult parts of a problem. They reached this conclusions

from an experiment where the terminal constraint weights from successful runs of SAPS were used as the initial weights for other runs. They found that the information encoded in the weights did not improve search performance in the second runs and in some cases it even took longer to solve some problems. They also argue, like Morris [81], that weights on constraints can have undesirable effects on the objective/cost landscape and as such there must be mechanisms to undo their impact.

The Guided Local Search (GLS) [113] for combinatorial search adopts a similar philosophy to the Breakout algorithm, but its emphasis is on problem features. Problem features can be subsets of a variable's domain or constraints. Penalties are attached to each problem feature, and those on features present[3] in a candidate solution are augmented in its cost function. Like the Breakout algorithm, penalties on those features present are increased whenever the local search is stuck at local optima. In GLS, penalties are increased proportional to their costs and the growth of penalties on a feature is controlled by a utility function which decreases over time. Penalties also change the shape landscape allowing the search to avoid regions containing "bad" features, but the impact of the penalties on the landscape is controlled by a lambda parameter which can be tuned to control the diversification / intensification bias of the search. GLS has been applied to and shown to be a competitive algorithm for problems in domains such as frequency planning [115], the travelling salesman problem [113], and function optimisation [114]. However, it still runs the same risks as the Breakout algorithm i.e. the potential for feature penalties to block paths to solutions is still present. Extensions to GLS proposed in [75], include an aspiration move where feature penalties are completely ignored, if there is a better solution in the neighbourhood of the current solution with respect to the original cost function. The extension also includes a probability for accepting random decisions.

In other related work, weights on variables have also been studied. In [88], the effects of dynamic variable weighing and continuous weight smoothing are investigated for tie-breaking for the variable selection heuristic in WalkSAT. Weights are associated with the number of times a variable's value has changed and where there is a tie for the next

[3]In the case that constraints are used as features, the constraints are present in a candidate solution if they are violated.

variable to be flipped, the variable with least weight is selected. The introduction of these schemes was shown to dramatically improve the performance of WalkSAT.

Unlike backtracking, local search algorithms are incomplete. There are no mechanisms for detecting that problems are unsolvable, nor are there any means to guarantee that solutions would be found even if they exist. Hence, in practice, local search algorithms are run with maximum time bounds or limits on the number of iterations. Furthermore, it has been demonstrated that local search techniques do suffer on structured problems [61] and are inferior to the inference of backtracking algorithms on such problems. Although, Hoos [54] argues to the contrary. Nevertheless, on some problem domains local search algorithms do outperform complete algorithms; they are used to solve larger problems within practical limits that are impossible with complete algorithms. For example, local search algorithms have been used to solve SAT problems with several hundred thousand variables, compared to a maximum of 600 variables for complete search algorithms [23].

Cooperative or parallel search for solving CSPs

According to Clearwater et al [19], the justification for cooperative search has amongst its many benefits performance speed ups in the time taken to find solutions and improvements in the quality of solutions generated. It is also argued that cooperative search with multiple search processes allows for exploration of more areas of the search space, either via different initial starting points, the use of different parameter settings, or different heuristics in the search processes. Resulting from these is the added advantage that cooperative search algorithms are likely to generate more unique solutions to problems.

We consider cooperative search to be generally about the exchange of information between search processes. With this view, we widen the umbrella of cooperative search to cover the host of population based heuristics in the literature. In cooperative search, a number of search processes run concurrently and periodically exchange information about profitable areas of the search space to exploit (or unprofitable areas of the search space to avoid). Information received from other processes may be used to: (i) resolve conflicts [52], where one process hits a dead-end; (ii) guide the search of multiple processes, each on

a different search strategy [74]. Information exchange in search can be explicit as already noted, or may be implicit in the nature of an algorithm. For example, in the evolutionary approach with Genetic Algorithms, a parallel search is carried out implicitly. Cooperation is enforced by selection and cross-over operators which allow the exchange of information about high fitness regions of the search space.

By viewing cooperative search as information exchange, the key issue with cooperative algorithms (or systems) will be determining what form of information is exchanged between search processes [112]. With genetic algorithms, partial solutions are exchanged. In [52] and [53], the explicit exchange of 'hints' between search processes is proposed, although the hints exchanged in the implementation presented were partial solutions. Agent's representing processes in the framework shared a common blackboard for storing and looking up hints to resolve conflicts in individual searches. In a similar approach, the work in [112] also proposed the exchange of tabu history in a group of concurrent tabu-search processes. The idea was to guide the search away from potential deadlocks. Milano and Roli [74] also proposed the exchange of partial solutions in a cooperative strategy for their framework, in which search with different heuristics (in some cases different parameters for search operators) run in parallel. The strategy adopted in the work was to take as input for a set of processes running population-based algorithms, the output of another set of processes running local search heuristics and vice versa. In work along similar lines presented in [17], several algorithms are run in parallel and at periodic intervals the overall best solution found by one of the algorithms used as new starting points for the other algorithms.

The definition of cooperation adopted in this study covers population based approaches for the reason that cooperation is implicit in the design and operation of such approaches. However, explicit cooperation mechanisms may still be incorporated into these approaches. The multi-population or parallel genetic algorithm [2] is an example of such. In this case, multiple populations are evolving, each on a separate 'island', and there is exchange of chromosomes between the populations at periodic intervals. Which, amongst other things, introduces some diversity into each sub-population and points the sub-populations to high fitness regions of the search space. Other approaches, such as the ant colony optimisation

algorithm [14, 109] and the particle swarm optimisation algorithm [59] use explicit markers from the most successful individuals in the search to guide the rest of the population.

2.1.2 Constraint propagation and consistency maintenance

According to Dechter [23], constraint propagation is perhaps the most fundamental concept in constraint reasoning. Constraint propagation and consistency maintenance are methods of inference that use information contained in constraints to rule out parts of a problem that can not be a part of a valid solution to the problem. These ideas have been explored in some of the earliest works in constraint reasoning (e.g. [118, 68]) and they still attract significant interest in the community.

Constraint propagation techniques are used to reduce the size of a problem; either by inferring new constraints from the combination of existing constraints or by deleting those values that do not appear in the set of allowed combinations in at least one constraint. Arc-consistency, which is probably the most popular of the constraint propagation techniques, does the latter. Pioneered in early works on constraint solving ([118] and [68]), arc-consistency works to ensure that each value in a variable's domain has at least one supporting value in the domains of other variables that are constrained with the variable. For example, take a CSP with two variables (x and y), where both variables have the same domain of 5 values $D_{x,y} = [1..5]$, and there is a single constraint $x > y$. Arc-consistency will delete 0 in x's domain since there are no values in y's domain that satisfy the constraint if $x = 0$; 5 is also deleted from y's domain in the same regard. This creates an equivalent problem [23] but with a smaller search space keeping just the values that can be present in a solution.

Depending on the nature of the constraints involved, arc-consistency techniques can detect that a problem has no solution when a variable's domain is completely deleted. But, doing this requires a complete evaluation of all value combinations in a CSP. This complete enumeration meant that the early arc-consistency algorithms (e.g. AC-1, AC-2, and AC-3 in [68]) were rather expensive, though in polynomial time, having worst case complexities of $O(ed^3)$; where e is the number of constraints in the problem and d is the

size of the largest variable domain. Later versions, such as AC3.1 [129], AC-4 [79], AC-6 [7], and AC-2001 [11], introduced measures to eliminate redundant checks and thus reduce the worst case complexity of to $O(ed^2)$.

Arc-consistency ensures that each constraint (or arc) in a CSP has at least one combination of values that satisfy it, but that can not in itself determine if the un-deleted values can be combined for a valid solution for a CSP. To deal with this, path-consistency [80] strengthens the propagation beyond the variables in the scope of one constraint i.e. determining if there are supporting values in other constraints for values satisfying an initial constraint. For example, a CSP with 3 variables (x, y, and z) and the constraints ($x < y, y > z$), is considered to be path consistent if for value in y's domain that satisfy the constraint ($x > y$) there is a support value in z's domain that satisfies the second constraint. Path consistency is also known as 3-consistency (for binary CSPs), but consistency algorithms that extend consistency propagation to larger sets of variables are known k-consistency algorithms. k-consistency, generalised in [31], determines that a CSP is k-consistent if for any consistent instantiations of k-1 variables, there is a value in the domain of an kth variable that satisfies all its constraints with the k-1 variables.

Despite the strengths of the constraint propagation techniques discussed here, they are rarely used alone to solve CSPs alone as there are times when search is still necessary and because the reduced problem space is still large. Therefore, constraint propagation techniques are often used in a pre-processing stage, or intertwined with search algorithms to improve search efficiency. For example, Arc-consistency algorithms can be combined with backtracking to maintain arc-consistency as variables are instantiated [96]; and this combination has been shown, in [10], to be the most efficient strategy for solving CSPs.

2.2 Distributed Constraint Satisfaction

Distributed problems of interest in this study typically come in the form of a number of decisions to be made individually by several participants involved in collaborative situations. There are a limited set of alternatives each participant can consider for each decision and there are restrictions on the alternatives that can be simultaneously selected for several

decisions. For example, in a Supply Chain system each company makes its own decisions on its production schedules and some of the choices made for these decisions are restricted by agreements and delivery decisions to be made by other participants in the chain. This example also highlights an important feature of distributed problems - all the information about the decisions and the alternatives for each decision are inherently distributed and can not be collected in a single location for problem solving.

In the seminal work for the field, Yokoo et al [122] extended the CSP framework to formally describe such distributed problems as Distributed Constraint Satisfaction Problems (DisCSPs) and introduced the idea of distributed approaches for solving them. In this formalism, distributed problems are solved by collections of automated software agents - each acting on behalf of a single participant in a problem. The decisions to made by the agents become variables in a problem and the alternatives for each decision become its variable's domain, while the restrictions on alternatives that can be selected are the constraints in the problem. In Chapter 3, we give formal definitions of DisCSPs and discuss some of the underlying assumptions of the formal models. However, in the following, we discuss algorithms for solving DisCSPs.

Algorithms for solving DisCSPs can be classified along similar lines with those for solving CSPs. Hence, the following review discusses backtracking, iterative improvement, and consistency propagation for DisCSPs. In addition, other methods that have no direct equivalents in centralised CSP solving such as those that use mediators to resolve deadlocks are also reviewed.

2.2.1 Distributed backtracking search

Just as in problem solving in single processor environments, backtracking search is a popular technique for distributed problem solving. The same theoretical guarantees of completeness and the amenability to analysis make it the most widely form of search studied in the DisCSP community. Direct extensions of centralised backtracking algorithms have been proposed for solving DisCSPs (e.g. Synchronous Backtracking [21]), but other such algorithms allow for asynchronous activity and introduce other measures to take advantage

of some of the peculiarities of problem solving in distributed environments.

In distributed backtracking, constraints are directed and as such agents agree on a static ordering so that values are proposed from the top to the bottom (via *ok*? messages), and the proposals are evaluated in the reverse direction. This way dead-ends are easily identifiable, i.e. when a set of proposals wipe out an agent's domain it can initiate backtracking. Synchronous Backtracking (SBT), which is earliest form of distributed backtracking, is a direct extension of centralised backtracking search and it tries to replicate the exact behaviour from that form of search in distributed environments. So just as in the centralised case, solutions are extended one variable at a time - an agent at a time, and moving up the agent ordering when backtracking occurs. But this approach is generally considered to waste resources because agents at the bottom of the ordering are almost always idle; especially where there is a lot of backtracking in the middle of the ordering. Asynchronous Backtracking (ABT) [122] tries to overcome this limitation by allowing all agents to act concurrently and asynchronously, as its name implies.

In ABT, agents still maintain a total and static ordering but they are all active simultaneously, constantly sending proposals to neighbouring agents lower in the ordering and at the same time evaluating the proposals they receive. In addition, agents learn, during the search, about combinations of proposals that can not be part of a solution. When such combinations are found, they are used to create new constraints in the form of no-goods, to prevent their recurrence. No-goods are sent by the agents that generate them to the nearest agent in the no-good higher up in the ordering. Occasionally, the proposals causing a no-good can involve agents not originally linked in the constraint network. Therefore, agents receiving such no-goods create new links with those agents they are not connected to; and hence are able to evaluate the new constraint.

The static ordering of agents in ABT allows the algorithm to avoid infinite processing loops, despite the concurrent activity by all agents. The advantages of its asynchronous approach include reduced idle time and earlier detection of deadlocks e.g. where a proposal from the first agent in the ordering wipes out the domain of the last agent in the ordering. ABT has been shown to be sound and complete, as well being quicker to solve problems

than SBT.

Several modifications (or new versions) for ABT have been proposed, including for example [108] and [132] which all present new heuristics to allow agents to change their ordering asynchronously and dynamically during the search. In particular, Zivan and Meisels [132] showed that this reduces the runtime of the algorithm and the number of constraint checks carried out. In another modification to ABT [16], agents are made to switch back and forth between asynchronous and synchronous activity. The forward phase of the search is asynchronous and backtracking is synchronous - to reduce the amount of redundancies in ABT, whenever agents send no-good messages they wait for the responses to those messages before resuming their search. This was also shown to improve ABT's performance.

The Distributed Dynamic Backtracking (DisDB), [9, 8], inspired by dynamic backtracking [33] improves on ABT, especially reducing the number of messages exchanged between agents [131]. DisDB is somewhat similar to ABT, it is also asynchronous and agents also learn from no-goods. But, DisDB's space complexity is polynomial since no-goods are regularly discarded, and only one no-good is retained for each domain value. Nevertheless, DisDB is complete. In a detailed study of DisDB in [8], it was found that performance is not significantly impaired on problems to the left of the complexity peak if agents do not create new links with unconnected agents when no-goods are received. In other work, Meisels and Lavee [72] show how additional information can be included in no-goods to improve its performance.

A distributed version of Graph-Based Backjumping [22] was introduced as Distributed Dynamic Backjumping (DDBJ) [83] which also includes dynamic variable and value ordering. In the algorithm which is semi-asynchronous, agents run the forward solution extending phase and the backjumping phase concurrently; just as in ABT and DisDB. But, the forward phase is made sequential by having agents send *ok?* and *Forward Check* messages rather than the lone *ok?* message in ABT. These synchronise the forward phase and allow the search to perform forward checking. DDBJ is complete and it has much lower space requirements than DisDB since agents do not store no-goods. Empirical evaluations

in [83] show that DDBJ outperformed DisDB and the Asynchronous Forward Checking (AFC) algorithm [73] on random problems.

The Asynchronous Aggregation Search (AAS) [107] is an ABT-like algorithm for a DisCSP where agents can jointly own variables; this is useful for problems like negotiation or those that involve joint decision making. Constraints are private to agents in that model, therefore in AAS rather than propose values agents send all the values for a variable consistent with their constraints.

Asynchronous activity in backtracking search is exploited further in the Interleaved Distributed Intelligent Backtracking (IDIBT) [41] where agents are able to run multiple searches in parallel. The idea is to use what would be agents' idle time to explore different regions of the search space simultaneously. In IDIBT, agents create a static ordering that results in one agent being designated as a Source agent. The source agent partitions the search space by creating search contexts for disjoint subsets of its domain. Search is initiated in each context when the source agent sends its first proposal in that context. As all messages include the context identification, agents are able to maintain information for the different contexts, permitting them to evaluate proposals and extend different solutions in parallel using an extension of Graph Based Backjumping. Like ABT and DisDB, IDIBT is also complete but it has a much lower space complexity. In later work by its author [42], additional heuristics were introduced to allow agents to maintain partial directional consistency during in a search with Conflict Directed Backjumping (CBJ) [89]. Similarly, the effects of Forward Checking and Look Ahead strategies on IDIBT were studied in [94]. These were found to reduce the number of constraint checks performed and the local computation efforts of agents. Distributed parallel backtracking search was also explored in Concurrent Backtracking (ConBT) [131] where agents retain partial solutions of different searches in parallel.

The idea of concurrent searches was extended further in the Multi-Directional Distributed Search [95] where different search algorithms can be run in parallel and useful information is shared between the searches. The authors argued that idle time is significant in distributed backtracking, and as such agents can use that time to solve the same

problem with a second algorithm. In their work, modified versions of ABT and IDIBT were run in parallel, each with different agent orderings. The experimental evaluation showed that the combined searches cut idle time significantly and provided a speed up of one order of magnitude compared to using either algorithm on its own.

2.2.2 Asynchronous Weak Commitment Search

A drawback of asynchronous backtracking is that when the agents highest in the ordering make bad decisions, agents below them have to exhaustively search all possibilities before the bad decisions are revised. Asynchronous Weak Commitment Search (AWCS) [121] tries to avoid this by allowing agents to dynamically change their positions in the ordering during the search. In addition, constraints are not directed in AWCS and as such, the Min-Conflicts [77] heuristic is used for value ordering.

Each agent in AWCS holds a non-negative integer value to represent its priority, and ties are broken in favour of the agents with the lowest lexicographic IDs. These priority values are also exchanged when agents communicate with their neighbours. During the search, each agent finds all values in its domain that satisfy all constraints with higher priority neighbours, and from these values it selects the value that minimises constraint violations with lower priority neighbours. Where an agent's domain has been wiped out by values of higher priority neighbours, it generates a new constraint in the form an undirected no-good and it sends the no-good to all higher priority neighbours involved in it. At the same time, the agent increases its priority to the highest in its neighbourhood. Storage of no-goods guarantee the algorithm's completeness, since there is a finite number of them and AWCS can ascertain that a problem is unsolvable if an empty no-good is created.

AWCS is efficient in terms of the number of asynchronous cycles performed, but it has the drawback of possibly creating an exponential number of no-goods during a search [49]. As each no-good is a new constraint, the number of constraint evaluations will increase as the search progresses. These drawbacks may limit the applicability of AWCS to small or loosely constrained problems [72].

In modifications to AWCS, Hirayama and Yokoo considered new learning schemes for generating smaller and more efficient no-goods [47] and Zhou et al have used constraint violations to determine agent prioritise [130]. AWCS has also been extended for agents holding multiple local variables in [124].

2.2.3 Distributed consistency maintenance techniques

Consistency maintenance in DisCSP solving is a challenge because of the privacy requirements/assumptions in the basic DisCSP model. Agents can not (are not expected to) know the full domain of variables owned by other agents; as such, the conventional methods for consistency maintenance or even for achieving arc-consistency can not be directly applied without relaxing the privacy assumption. Hence, some of the early work in distributed consistency maintenance more or less ignored the privacy assumption and focused on how arc-consistency can be achieved on multi-processor environments [90] or on improving the efficiency of the process [82, 40, 93]. Other work (discussed next) departs from the basic DisCSP model, highlighting scenarios where the basic model is not sufficient and therefore introduce new algorithms for achieving arc-consistency without violating the privacy requirements of the underlying models.

The DisCSP model adopted in [107] considers scenarios like distributed negotiation, where variables are jointly owned by agents. Therefore domains are public knowledge but constraints are private. As a result, the Asynchronous Aggregation Search, presented in that work, allows agents to propagate constraints and exchange partial solutions during a search. In their work on the Distribute Forward Checking (DFC) [15] algorithm, Brito and Meseguer consider a DisCSP model where domains and constraints are public knowledge, but the values agents select for their variables are private. During the search, which is intertwined with a distributed backtracking algorithm, agents propagate constraints by pruning domains of un-instantiated neighbours. So rather than sending their labels to neighbours, agents send a list of domain values available to neighbours.

Also working with a different model in [92], Ringwelski and Wallace introduce constraint agents in addition to agents representing variables. The constraint agents are

linked to and communicate with 'variable' agents. The role of the constraint agents was to propagate constraints that result from variable instantiations or constraint additions[4]. Like DFC, the consistency maintenance is intertwined with a search algorithm.

The Asynchronous Forward Checking (AFC) algorithm [73] works with the standard DisCSP model to combine synchronous backtracking search with asynchronous constraint propagation. Agents take turns to construct a solution sequentially and, at the same time, the owners of un-instantiated variables perform forward checking tasks notifying owners of instantiated variables of values that do not propagate.

2.2.4 Distributed iterative improvement

As we have discussed in Section 2.1.1, iterative improvement search (or local search) allows a search to start off with a complete assignment of values to variables and proceeds to iteratively move the state towards a valid zero-cost solution. Although methods based on local search are incomplete, they have the advantage of converging quicker to good quality solutions (or possibly zero cost solutions) than backtracking algorithms, especially on large problems. But, with this speed of convergence there is the ever present potential for such methods to converge to local optima. As such the challenge in the design of new iterative improvement algorithms is devising strategies for effectively dealing with local optima, that allow a search leave a plateau in the objective landscape and possibly prevent it from returning to that region.

In DisCSP solving, strategies considered for dealing with local optima have included landscape modification with constraint weights and randomisation to perturb solutions that cause a search to jump to other areas of the search space. The earliest work in the distributed iterative improvement is the Distributed Breakout Algorithm (DBA) [123, 49] which was inspired by the Breakout algorithm [81](see Section 2.1.1).

In DBA, agents carry out a distributed steepest descent search by exchanging possible improvements to a candidate solution and implementing the best improvements that do not conflict with each other. To do this, agents act concurrently alternating between the *improve* and *update* cycles. In the improve cycle, each agent finds the value in its

[4]They were working on dynamic DisCSPs that change as the solution process progress.

domain that minimises its weighted constraint violations and computes the improvement to its current assignment. These improvements are exchanged between the agents, and the agents with the best improvements are allowed to change their values in the update cycle. Ties are broken with the agents' lexicographic IDs in the case that two or more neighbouring agents have the same possible improvement.

Deadlocks at local optima are dealt with using weights attached to constraints and those on violated constraints are increased whenever agents are at deadlocks. To minimise communication costs, DBA's authors consider the notion of quasi-local-optima which is much weaker than local optima. It is described as a state in which a subset of connected agents can not find any improvements to their local evaluations. Agents individually increase weights, in the update cycle, on the constraints they violate when they detect that they are at quasi-local-minima. Hirayama and Yokoo [123] also investigated the effects of increasing weights at real local optima and found that performance in terms of the number of cycles taken is better. But this comes with the requirement that each agent is able to communicate with all other agents in the constraint network (even those it does not share constraints with) and this increases the number of messages exchanged between agents considerably. Empirical evaluation of DBA showed that it outperformed Asynchronous Weak Commitment Search on difficult problem instances; DBA solved more problems and it did so in less time [123].

Originally, it is assumed that each agent in DBA owns just one variable, but newer versions in [48, 25] extend the breakout method for problems where each agent owns multiple local variables. Other work in [119], introduce some randomisation for tie-breaking in DBA and extend it for handling distributed constraint optimisation problems.

In the Distributed Stochastic Algorithm (DSA) [27, 28, 128], the authors introduce a stochastic light-weight strategy for dealing with local optima in their work on distributed target tracking. Agents act synchronously and in parallel, each selecting a value minimising the number of constraints violated given assignments received from neighbours in the previous iteration. To reduce incoherency and to avoid local optima, agents have random activations such that in each iteration an agent decides with a fixed probability to retain

its current assignment. DSA was shown to converge quicker to local optima than DBA on distributed scan scheduling problems - cast as distributed constraint optimisation problems. But, Hirayama and Yokoo [49] point out that once DSA is stuck at a local optima it has no means of escaping; and as such it would not be strong in decision problems where the objective is to satisfy all constraints.

Arshad and Silaghi considered improvements to DSA and extend the framework in Distributed Simulated Annealing (DSAN) [5], where they introduce additional random decisions to allow agents to occasionally select values that may not improve their evaluations. Inspired by the Simulated Annealing [60], DSAN's authors introduce an additional parameter to control the probability of making the non-improving changes and they allow this parameter to decay over time. Thus, the search is able to explore more regions early in the process and it increasingly scrutinises good regions later on. DSAN was shown to outperform DSA in the experiments reported in [5], but like the works on DSA they also focus on finding good quality solutions quickly rather than zero cost solutions.

2.2.5 Third party mediators

The common thread in all the work reviewed in this section so far is the notion of agents acting 'independently' to either extend solutions or to resolve conflicts. But given the limited view each agent has of the problem, they can not fully evaluate the impact of the decisions they make on other parts of the constraint network. For example, there is the risk that actions taken to resolve a conflict by one agent in one part of the DisCSP will cause the appearance of a new conflict in another part of the problem. To overcome this, a number of approaches in the literature propose the referral of conflicts to third parties or *mediators* for resolution.

Mailler introduced a cooperative mediation protocol in the Asynchronous Partial Overlay (APO) algorithm [70], where agents can opt to mediate conflicts for their neighbours. In this hybrid of centralised and distributed search, when an agent is mediating a conflict, it requests for information from its neighbours about the variables and constraints in the conflict (which is outside its view of the problem) so that it can fully anticipate the impact

of its decisions. A centralised systematic algorithm is used to resolve the conflict with the collected information. Mediating agents can detect the absence of solutions, and hence the APO algorithms are certified to be complete. In his empirical evaluations, Mailler showed that APO outperformed AWCS in distributed graph colouring and distributed target tracking problems where it typically required fewer iterations to find solutions. A similar approach for mediation was presented in [120] where virtual agents were constantly created and used to mediate in conflicts.

Sathi and Fox also present a hybrid of centralised and distributed search in [97] for their work on resource re-allocation. In their approach, a central mediator has a global view of the problem and resolves conflicts that depend on multiple resource offerings. A somewhat related approach in [4] utilises a separate agent acting as a centralised no-good processor. The algorithm requires that agents regularly check with the no-good processor before assigning values to their respective variables. While in the broker model presented in [65], agents routinely send unresolved parts of their local sub-problems to a central mediator for resolution.

2.2.6 Anchor agents

While third party mediators are used to resolve conflicts during a search, the notion of an anchor agent is one that is predetermined to be central (or a backbone) of a DisCSP. Anchor agents propose partial solutions for other agents to extend, or they can act as mediators when partial solutions are not unanimously consistent.

In their work on job shop scheduling, Liu and Sycara proposed the Anchor and Ascend algorithm [67], where agents controlling bottleneck resources are designated anchor agents. Anchor agents will first seek to optimise their local sub-problems and modify the solutions when they are proven to be infeasible. Two variations on this theme are considered in [110]. In the first variation, a central agent attempts to find values for all variables in its sub-problem (with emphasis on variables involved in inter-agent constraints). Following which, other agents search for consistent sub-solutions. The second variation, is a direct reversal of roles where the central agent awaits the sub-solutions before attempting to solve

its sub-problem. The second approach was shown to be more efficient, as it eliminates the amount of wasted search effort by peripheral agents that takes place when a single agent is not able to find consistency with the initial proposal.

Chapter 3

Problem Formalisation

In this chapter, we present a formal description of the DisCSPs model used for this work (Section 3.1). We discuss the assumptions made about the nature of the problems in Section 3.2 and highlight the scope of the formal model (and the study) in Section 3.3.

3.1 The Distributed Constraint Satisfaction Problem - DisCSP

A DisCSP is formally described as $DisCSP = \langle A, X, D, C \rangle$ where:

- $\mathbf{A} = \{a_1, a_2, ...a_m\}$ is a set of m agents.
- $\mathbf{X} = \{x_1, x_2, ...x_n\}$ is a set of n variables.
- $\mathbf{D} = \{D_1, D_2, ...D_n\}$ is the corresponding set of domains for each x_i; i.e. each domain (D_i) is a finite set of discrete values that can be assigned to x_i.
- $\mathbf{C} = \{c_1, c_2, ...c_p\}$ is a set of p constraints that limit values that can be simultaneously be assigned to the variables in the scope of c_i.

There is a distribution of variables to agents, such that each variable belongs to exactly one agent while each agent may represent more than one variable i.e. each decision in a collaborative problem can only be made by one participant but each participant can make several decisions.

Definition 3.1 (Neighbours) *Two variables are neighbours if they are both in the scope of a constraint. Two agents are neighbours if at least one pair of variables they own are neighbours.*

Definition 3.2 (Neighbourhood) *An agent's neighbourhood is the set of its neighbours i.e* $N = \{x_1, x_2, ...x_k\}$.

Definition 3.3 (Inter-agent constraint) *An inter-agent constraint is a constraint between variables owned by different agents.*

Definition 3.4 (Intra-agent constraint) *An intra-agent constraint is a constraint between variables owned by the same agent.*

Definition 3.5 ($AgentView_t$) *An AgentView is the set of assignments* $\{x_1 = v_1, ...x_k = v_k\}$ *for variables belonging to agent's neighbours at time t.*

Definition 3.6 (No-good) *A no-good is an inconsistent AgentView i.e. as a result of neighbours' current assignments there is no value in domain of a variable that satisfies all constraints attached to it.*

Agents try to solve a DisCSP by exchanging value assignments and other algorithm specific information. For example, in distributed backtracking agents can infer new constraints during a search and exchange them as well. They determine that a DisCSP is solved when they simultaneously hold values for their variables that satisfy all the constraints in a problem.

3.2 Privacy requirements and assumptions

3.2.1 Privacy and limited information

The distinguishing features of a DisCSP are privacy and limited availability of information. Yokoo et al [122] cite these as the key reasons why DisCSPs have to be solved by distributed techniques and as the main challenge of this approach. Privacy, and security of information, requires that information about a problem remains in the hands of its

owners. Hence, the problem information can not be collected in one location for solving with a centralised technique. This requirement also distinguishes distributed constraint solving from parallel (e.g. [62, 126]) or cooperative (e.g. [19]) approaches to problem solving, which rely on a constant flow of information between several search processes to improve efficiency.

Generally, privacy is related to variables' domains. Each agent is expected to keep the full domain of a variable it owns private; therefore an agent can not know the full domain of a variable it does not own. Agents are also expected to keep information about constraints they are involved in private. This is illustrated with the example in Figure 3.1, which uses the standard notation to depict DisCSPs as graphs. In this diagram, the nodes represent variables and arcs connecting nodes indicate the presence of constraints between variables. In the example, variable x_1 has separate constraints with variables x_2, x_3, and x_4; but privacy requirements prevent the agent that owns variable x_1 from being aware of the constraint between variables x_3 and x_4. Agents are not permitted to reveal any information to a neighbour about constraints with other agents. Hence, each agent always has a partial view of the problem being solved.

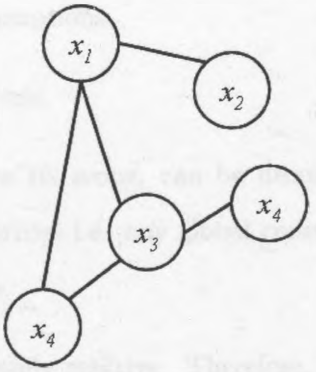

Figure 3.1: Privacy of constraints in Distributed Constraint Satisfaction Problems.

Other forms of privacy have been considered in the literature. For example, Silaghi et al [107] argue that the standard model of privacy may not be suitable for problems such as distributed negotiation. Therefore, they propose an alternative model where variables and domains are public knowledge while constraints remain private. In some other work, Brito and Meseguer [15] propose a model in which variables' domains are public knowledge but

actual values assigned to variables remain private. They also considered Partially Known Constraints (PKC) (or asymmetric constraints) in which each agent in the scope of the constraint is aware of just a handful of the forbidden combinations in the constraint.

In this work, we continue with the standard notions of privacy keeping variables' domains and information about unrelated constraints private. We assume that each agent has a partial view of a problem, and it is only aware of the constraints it is directly involved in. We also assume that preferences for particular values (if any exist) are also private. We restrict the amount of information agents are allowed to reveal during the problem solving process to the values currently assigned to their respective variables. As such, agents may not inform their neighbours about values received from other agents. In the case where each agent has multiple variables, an agent can not inform its neighbours about its local variables not sharing constraints with those neighbours. This violates the privacy requirement if neighbours can use such information to infer additional details about agents, and where such inference is not possible the information is redundant.

3.2.2 Other assumptions in the model

We also make the following assumptions:

- All constraints are symmetric.
- Any constraint with X as its scope, can be decomposed into an aggregation of constraints with smaller arities i.e. any global constraints can be decomposed into several smaller constraints.
- Agents' behaviours are purely reactive. Therefore, they can not make predictions about future states and they base all their decisions on current (or past) information.
- Agents communicate with each other by sending messages with the values assigned to their variables as well as requests to neighbours to perform certain actions (clarified in Chapters 5 to 7). We assume that agents can only communicate with their neighbours.

- Communication is reliable i.e. messages will get to all recipients with a fixed finite delay. We also assume that all messages are received in the order in which they were sent. However, in Section 5.7, this assumption is relaxed when the impact of unreliable communication is studied.

3.3 Scope of the model

For the DisCSPs we study in this work, we focus on static versions where the components of a problem i.e. agents, variables, domains and constraints, are specified in advance before agents attempt to solve it. These components remain fixed during the problem solving process. There are no additions or modifications to the components. In addition, the aim of the adopted model is to find the first full assignment of values to variables that satisfies all constraints simultaneously. Therefore, this does not extend to Distributed Constraint Optimisation Problems (DisCOPs) [78] where the objective is to find the best solution, from all possible solutions to a problem, or to optimise a given objective function.

Chapter 4

Penalty-based Strategies for Dealing with Local Optima

4.1 Introduction

Solving problems with iterative improvement/local search algorithms offers the advantage of quicker convergence over constructive search. However, this benefit comes with a potential for convergence to locally optimal non-solution states and/or a propensity for the search to wander about on sub-optimal plateaus in the objective landscape. Strategies proposed to overcome these drawbacks typically introduce some non-improving decisions with the intention of moving the search away to other, possibly unexplored, regions of the search space in order to resume the search for a solution. Alternatively, some strategies try to determine the sources of the deadlocks associated with the local optima and seek moves that directly attempt to resolve them. A widely studied approach for doing this, as highlighted in Section 2.1.1, modifies the shape of the objective/cost landscape with weights attached to constraints. Algorithms based on this approach have been shown to be remarkably effective for solving SAT problems in centralised settings.

In Section 4.2, we reconsider landscape modification with constraint weights and highlight a key weakness of the approach. Following that, in Section 4.3, a new approach for modifying objective landscapes with penalties attached to domain values is introduced

along with an illustration of how this strategy overcomes the identified weakness of using constraint weights. A similar penalty based strategy for solution perturbation is introduced in Section 4.4. Both strategies, which are discussed in detail, form the basis of new distributed constraint satisfaction algorithms introduced in Chapters 5, 6, and 7 of this thesis.

4.2 Modifying the cost landscape with constraint weights

The idea of dealing with local optima by modifying the shape of the cost landscape was introduced in a modification to local search by Morris [81] in his work on local search for boolean satisfiability. The aim was to provide a mechanism to allow a search focus its efforts on resolving clauses that were repeatedly unsatisfied and regularly associated with local optima. Hence, weights were attached to clauses (or to constraints in the case of CSPs) and the cost function of the problem to be solved was modified as follows:

$$h = \sum_{i=1}^{n} cw_i * viol(c_i) \qquad (1)$$

where:

c_i is the ith constraint

cw_i is the weight of the ith constraint

$viol(c_i)$ is 0 if the constraint is satisfied, otherwise it is 1

The weighted sum of violated constraints is used to evaluate candidate solutions. The weights attached to violated constraints are increased whenever the search is stuck at local minima to change the shape of the cost landscape. This drives the search away from the deadlocked region and, at the same time, it should have the effect of blocking out other regions where solutions do not exist.

However, we argue that modifying landscapes this way can sometimes be futile and it can cause a search to remain stuck at deadlocks on plateaus. The alterations in the

landscape caused by the constraint weight modifications affect the altitudes of plateaus but the plateaus remain flat. As such, there is no room for the search to find a path out of a local optimum.

To illustrate this, take the example DisCSP in Figure 4.1 and its resulting cost landscape in Figure 4.2. Given the current assignments, the search is in a deadlock state (point P in region A) violating constraint c_1 - on a plateau in the cost landscape. There are two other plateaus in the landscape: region B where both constraints are violated and region C where just c_2 is violated; and a small region with solutions (e.g. $x = 1, 10 \geq y \leq 12$).

Variables	$[\ x,\ y\ ]$
Domains	$D_x = [1..20] \qquad D_y = [1..20]$
Constraints	$[\ c_1 = \{\ y - x > 9\ \},\ c_2 = \{\ x + 3y < 40\ \}\ ]$
Constraint weights	$cw_1 = 1$ and $cw_2 = 1$
Current assignments	$x = 8,\ y = 4$

Figure 4.1: An Example DisCSP

To resolve the deadlock using constraint weights, the weight of the violated constraint is increased to ($cw_1 = 2$) and this results in the modified landscape in Figure 4.3(a). While the altitudes have changed, the plateau around the deadlock remains, as well as the other plateaus in regions B and C. Increasing cw_1 further results in the landscape in Figure 4.3(b), the plateaus are still intact and therefore the search is unable to find a path out of the plateau and the deadlock remains unresolved.

Recent work on constraint weighted local search (e.g. [29, 30, 111]) have arguments against allowing weights to grow unbounded in their use for resolving conflicts at local optima. Amongst other things, blocking possible paths to solutions is cited as a reason why the growth of weights should be controlled, and therefore propose weight decay schemes to limit any detrimental impacts of weights on cost landscapes. However, we argue that the introduction of weight decays may not necessarily improve the effectiveness of using constraint weights to contorting plateaus. We argue that the effects of weight decays only serve to change the altitudes of plateaus (i.e. pushing them down) just as they are changed when weights are initially increased.

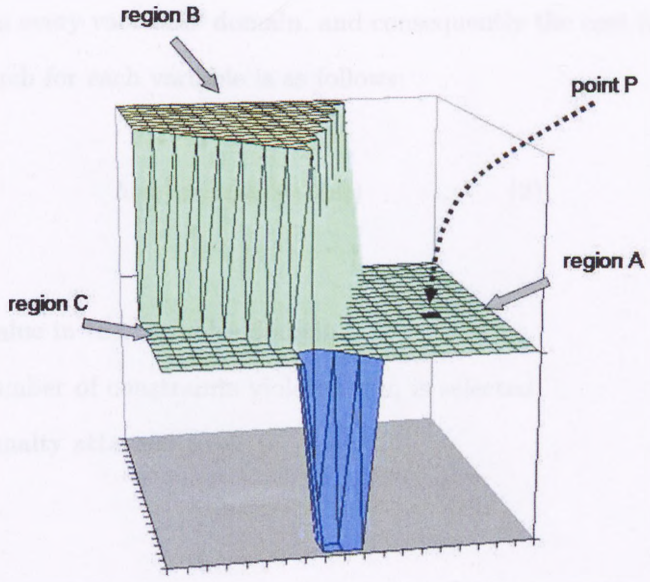

Figure 4.2: Cost landscape for the DisCSP in Figure 4.1.

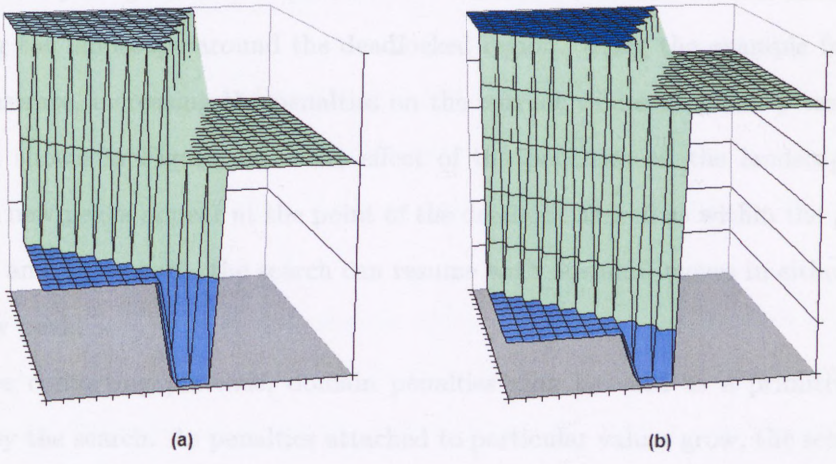

Figure 4.3: Effect of constraint weight modifications on the cost landscape for the DisCSP in Figure 4.1; (a) $cw_1 = 2$ and (b) $cw_1 = 5$.

4.3 Modifying cost landscapes with penalties on domain values

Given the highlighted weakness of using constraint weights to deal with local optima, we introduce a new penalty based strategy as an alternative mechanism. This new approach is finer-grained and shifts emphasis from constraints violated at deadlock states to the assignments associated with those violations. Therefore, a penalty is attached to each

individual value in every variables' domain, and consequently the cost function to be minimised by the search for each variable is as follows:

$$h(d_i) = v(d_i) + p(d_i) \qquad (2)$$

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 penalty attached to d_i

When the underlying search is stuck at a local optimum, penalties attached to the values currently assigned to the variables with violated constraints are increased, therefore contorting the landscape around the deadlocked region. Using the example from Figure 4.2 to illustrate, increasing the penalties on the current values of x and y results in the landscape shown in Figure 4.4. The effect of the penalties on the landscape is more dramatic; new peaks appear at the point of the deadlock as well as within the plateaus in regions B and C. As such, the search can resume with downhill moves in either direction of the new peak.

Besides contorting plateaus, domain penalties may be used as a primitive form of learning by the search. As penalties attached to particular values grow, the search is able to gradually "learn" of the association between the assignments and local optima. Hence, regions containing those assignments are excluded from further exploration as the search progresses.

The domain penalties introduced here are somewhat similar to the feature penalties in the Guided Local Search (GLS) algorithm [113] but differ fundamentally in the way they are used. First of all, in GLS solution features are penalised rather than individual domain values as we suggest. For GLS, solution features are properties of a solution that can be used to define it such that all features can not appear in all solutions at the same time. The choice of solution features are dependent on the problems being solved. For

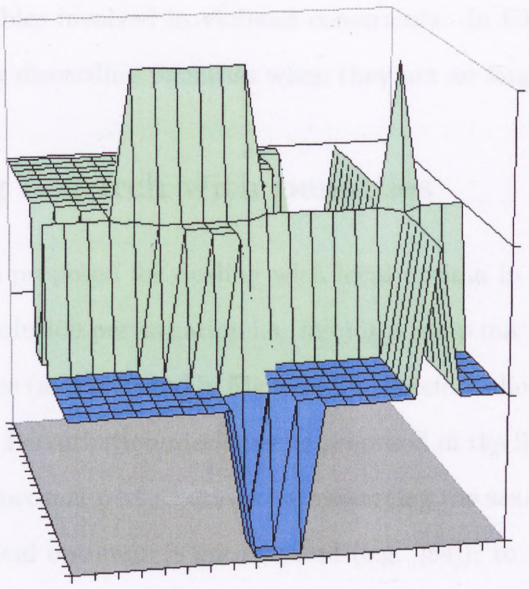

Figure 4.4: Effect of penalty increases on the cost landscape for the DisCSP in Figure 4.1 i.e. increased penalties attached to $D_x(8)$ and $D_y(4)$ from 0 to 1.

example, location-facility pairs were used as features in GLS for the Quadratic Assignment Problem, while jobs were used as features in the version for work force scheduling, and non-overlapping intervals of variables' domains were used as features when GLS was applied to function optimisation.

When a search is stuck at local optima in GLS, the cost of the features present at the local optima as well as the number of times the features have been penalised are used to determine the utility for penalising the feature $(Util(f_i))$[1]. And f_i with the highest utility is penalised. However, $Util(f_i)$ decreases over time the more f_i penalised, giving GLS room to penalise other features. Obviously, feature penalties are included in a problem's cost function and there is an additional regularisation parameter that is used to control the impact of penalties on the cost function - and by extension the explorative behaviour of the search.

As we mentioned, we propose simple penalties: one for each value in each variable's domain. So, our "feature" set is fixed irrespective of the problem type. The increments when the search is stuck are additive, and applied to the penalties on values currently

[1]Here we use f_i to refer to the ith feature.

assigned to those variables involved in violated constraints. In Chapter 5, we introduce additional heuristics for discarding penalties when they are no longer needed.

4.4 Perturbing a search with penalties

Of the many strategies proposed for dealing with local optima in iterative improvement search, the simplest is solution perturbation i.e. forcing a jump out of a plateau to another region in the search space (as illustrated in Figure 4.5), therefore allowing the resumption of intensification activity. Perturbation mechanisms proposed in the literature come in many forms and range from maximal perturbations i.e. restarting the search with a new random instantiation when a local optimum is encountered (e.g. [54]); to minimal perturbations which try to minimise deviations from existing search trajectories in an attempt to preserve much of the previous search effort up to the deadlocked state. Examples include Iterated Local Search [86] where random values are assigned to some variables, random walks in SAT solvers [102], and the mutation operator in genetic algorithms.

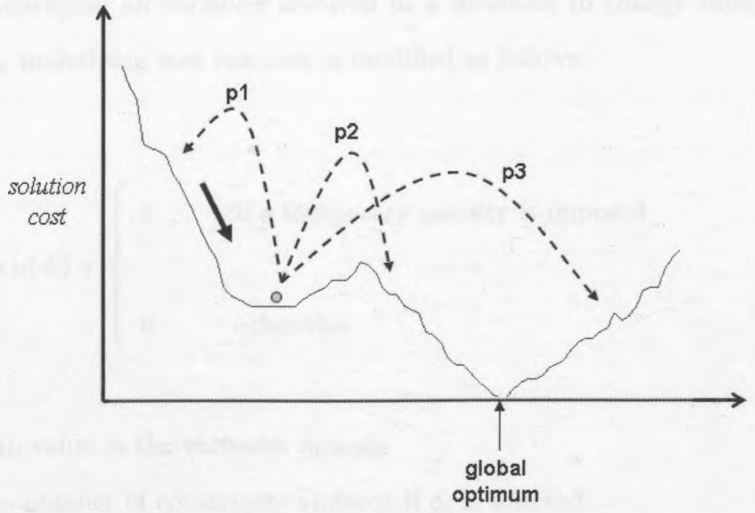

Figure 4.5: Schematic illustration of solution perturbations - p1, p2, and p3.

Minimal perturbations have been shown to be effective in dealing with local optima, especially in the SAT domain, but their limitations include a potential for a search to expend a lot of effort wandering around a plateau before finding a way out. Where perturbations are too weak, the search is unable to leave the deadlock state. For example,

in experiments evaluating the effects of randomly perturbing deadlocked variables in a simple greedy hill-climber (outlined in Figure 4.1), we found that the perturbations had no effect 67% of the time. In addition, some perturbations may push a search back up its trajectory (e.g. p1 in in Figure 4.5), and can possibly result in infinite oscillation between a set of points in the search space.

Algorithm 4.1 Greedy sequential hill-climber.

```
1: initialised problem with random value assignments
2: while solution not found do
3:    for each xi in X do
4:       select value d from D(x_i) that minimises constraint violations
5:    end for
6:    if no changes made then
7:       apply perturbation
8:    end if
9: end while
```

Building on the ideas on landscape modification presented in Section 4.3, a new penalty-based perturbation strategy is introduced in this work. The idea is to induce jumps by encouraging all variables involved in a deadlock to change their assignments. Therefore, the underlying cost function is modified as follows:

$$h(d_i) = v(d_i) + \begin{cases} t & \text{if a temporary penalty is imposed} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

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

t is the temporary penalty ($t > 1$)

The temporary penalty is imposed on current values assigned to deadlocked variables, therefore worsening the evaluation of those assignments. As a result, a new value that

minimises equation (3) is assigned to each variable; following which the penalty is discarded. This approach allows us to combine intensification and diversification in a single move because by selecting a value that minimises the sum of penalties and violations the search is pushed, but not too far away, from its existing trajectory.

To evaluate the effectiveness of the perturbation scheme, an experiment was devised whereby a simple greedy hill-climber (Algorithm 4.1) is driven to a local optimum and then perturbed with different heuristics. The effects of the perturbations were evaluated when the algorithm either found a solution or settled on a new local optimum. Of particular interest, were the number of deadlocks resolved, and critically, the number of new constraint violations caused i.e. the number of previously satisfied constraints that become violated. These were evaluated on binary DisCSPs in which all constraints were linear inequalities between pairs of variables.

We considered two types of penalty based perturbations: one in which the earliest variable involved in the deadlock is perturbed and one in which all variables involved in the deadlock are perturbed. These strategies were compared with random perturbations: (i) unilateral perturbations where a random value is selected for one variable involved in a deadlock, and (ii) multilateral perturbations where random values are selected for all variables in a deadlock. There were also comparisons with perturbations using temporary constraint weights. The results are summarized in Table 4.1.

Perturbation strategy	% of constraints resolved	% of times new violations were caused
Random (unilateral)	27	32
Random (multilateral)	50	42
Temporary constraint weights	84	66
Temporary penalties (unilateral)	39	37
Temporary penalties (multilateral)	57	43

Table 4.1: Effect of perturbation strategies with the greedy hill climber. Tested with 100 DisCSPs $\langle n = 30, d = 5, p1 = 0.3 \rangle$.

The results show that the advantage of constraint based perturbation is that it resolves a high percentage of constraint violations that cause deadlocks, but the drawback is that in doing so deadlocks are transferred to other parts of the constraint graph. Uni-

lateral random perturbations are worse on both metrics; very few constraints are resolved and at the same time, more constraint violations are caused. Perturbing with unilateral temporary penalties also results in similar outcomes.

It is clear that both multilateral perturbation schemes perform equally well on both metrics, and have the clear advantage of not transferring as many deadlocks to other parts of the constraint graph. But, between them, the temporary penalty strategy is more efficient. With random perturbations, within this framework, both perturbation and intensification can not be combined in a single iteration i.e. one can not select a random value for a variable and immediately select a value minimising the number of constraint violations. This returns the original value and hence the effects of the perturbation are negated immediately. Therefore to make random perturbations work, the deadlocked variables are perturbed in one iteration and the response to these can only start in the succeeding iteration.

When combined, both metrics in Table 4.1 suggest that the net effect of the temporary constraint weights make it the overall best i.e. the lowest sum of the percentage of unresolved constraints and the percentage of the number of times new violations were caused. However, on closer scrutiny of the results, we find that for every five constraints resolved, the strategy with constraint weights causes two previously satisfied constraints to become violated, compared to one new violation for every five resolutions using the temporary penalties. And on that basis, we consider perturbation with the temporary penalty as a competitive alternative.

4.5 Chapter Summary

Two penalty based strategies for dealing with local optima in iterative improvement search were introduced in this chapter. In the first strategy, deadlocks are dealt with by modifying the cost landscape with penalties on domain values. We showed that this approach has a more profound impact on cost landscapes, compared to similar constraint based approaches, hence we argue that it is more effective at resolving deadlocks. The second strategy introduced was a penalty based mechanism for search perturbation. We showed

that, compared to some other perturbation heuristics, the new approach has the advantage of resolving existing deadlocks while not causing as many new deadlocks in other parts of the constraint graph.

Chapter 5

Distributed Penalty Driven Search

5.1 Introduction

In this chapter, we introduce Distributed Penalty Driven Search (DisPeL) for solving DisCSPs where each agent owns just one variable. DisPeL is a distributed iterative improvement search algorithm that deals with local optima using the penalty based strategies introduced in Chapter 4. We discuss the algorithm and its behaviour, and present results of its empirical evaluation along with comparisons with the Distributed Breakout Algorithm which is based on a somewhat similar philosophy with DisPeL.

This chapter is structured as follows. The new algorithm, DisPeL, is introduced in Section 5.2, and in Sections 5.3 and 5.4, parts of the algorithm's strategy are discussed in detail and their impact on the algorithm's overall behaviour examined; while its theoretical properties are discussed in Section 5.5. Results of empirical evaluations are presented in Section 5.6, and the effects of unreliable communications on DisPeL's behaviour are discussed in Section 5.7.

5.2 Distributed Penalty Driven Search (DisPeL)

5.2.1 Overview

DisPeL is designed to solve DisCSPs where each agent controls just one variable[1] and the objective is to find the first solution that satisfies all constraints simultaneously. It is a greedy hill-climber, or an iterative improvement algorithm, in which agents, in a fixed ordering, take turns to improve a random initialisation. Therefore, unlike conventional hill-climbing, it is an approach that accepts sequential improvements in each iteration rather than the best possible improvements (building on Algorithm 4.1). Determining the best improvement to implement in the conventional way has significant cost implications in distributed problem solving; it requires each agent to compute a possible improvement and all agents collectively determine which improvements to implement by exchanging the computed values. However, using sequential improvements communication costs are reduced as all improvements are accepted and the information, used to make decisions, is always coherent.

The core of DisPeL's strategy is its use of penalties to modify underlying cost landscapes in order to deal with local deadlocks that prevent agents from improving the solution. 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 temporary penalties in an attempt to force agents to try other combinations of values, and allow exploration of other areas of the search space.
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.

Penalties are used collaboratively, so that whenever an agent detects a deadlock and has to use a penalty, it implements the penalty on its current assignment and asks its

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

neighbours to implement the same penalty on their current assignments as well. 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 to initiate when a deadlock is encountered.

We assume that all constraints are undirected, therefore each agent in DisPeL will evaluate, locally, all constraints attached to its variable. Hence, each agent will communicate, in a synchronised manner, with all other agents that are co-constrained with it exchanging value assignments and requests to impose penalties.

5.2.2 Algorithm details

DisPeL is an iterative improvement algorithm in the sense that it starts off with a random flawed solution which agents take turns to improve until a valid zero cost solution is found. The search is generally downhill in the cost landscape (or uphill in the objective landscape) where, in each iteration, agents use the min-conflicts heuristic [77] (or local repair) to select values that minimise the number of constraints violated. The cost function for each agent is modified to include two types of penalties, so as to incorporate the penalty driven strategies for landscape modification and solution perturbation, 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

$v(d_i)$ 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 ($t = 3$) [2].

As such, when it is an agent's turn to improve the solution state, it always selects the

[2]This value may be optimised for different problem classes, however, unless explictly stated we use this value for t in all our experiments irrespective of the problem size.

least cost value in its domain i.e. the value with the lowest sum of constraint violations (given current assignments to neighbouring variables) and penalties. Penalties are imposed (in the case of the temporary penalty) or increased (in the case of incremental penalties) when a deadlock is encountered as follows:

- If it is the first visit to a particular deadlock state, the earliest agent involved in the deadlock imposes a temporary penalty on its current assignment and sends a message to lower priority neighbouring agents involved in the deadlock to impose temporary penalties on their assignments when it gets to their turns to act. Each agent discards the temporary penalty on the domain value as soon as it is used.
- If the deadlock state is being revisited, the earliest agent involved increases the incremental penalty attached to its current assignment and sends a message to all its lower priority neighbours to do the same. Note that the incremental penalties are not discarded after being used.

We assume that deadlocks occur at quasi-local-optima rather than at real local optima - the cost of detecting if the whole solution is stuck at a real local optimum is high and doing this repeatedly can increase the search costs of the underlying algorithm exponentially. We determine that an agent is at a quasi-local-optimum when its *AgentView* (see Definition 3.5) is unchanged in two successive iterations and its current assignment is inconsistent. This differs slightly from the original definition of quasi-local-optima (in [123]) as states where agents have no values in their domains that reduce the number of constraints violated. As we discuss in Section 5.3, agents need to wait for the additional iteration to ascertain the deadlocks and prevent the use of penalties to resolve deadlocks that may not exist.

The temporary penalty, used when deadlocks are first encountered, is used to perturb the solution so that agents are forced to explore other regions of the search space to find combinations of values that can resolve the deadlocks. But it is used in a way that tries to induce a localised perturbation, which is intended to keep the search in nearby regions of the search space. Therefore, when an agent uses a temporary penalty, it will make a

request to those lower priority agents violating constraints with it to also use the penalty, thus making implicit requests for them to try alternative values that could possibly resolve the conflict - even though the initiating agent is also changing its value. If the attempt is unsuccessful, and the agents return to their assignments at the deadlock, incremental penalties attached to the current assignments are increased. But this time, the agent initiating the resolution will request that **all** its lower priority neighbours increase the value of the incremental penalties attached to their current assignments. The idea is that the combination of current assignments prevents one or more agents from finding consistent values and can not be part of a solution; and as such the increased costs (h) make the assignments less attractive and prompt exploration of other areas of the search space.

Incremental penalties also serve as a primitive form of short-term learning (or search memory) which allows agents to learn to avoid selecting values repeatedly associated with deadlocks. These penalties are set to zero at initialisation, and are increased during the search. A side effect of using incremental penalties to resolve deadlocks is the potential for the penalties to dominate agents' cost functions, since penalties are given equal weighing with constraint violations. This domination can drive the search away from promising regions of the search space, because as penalties grow, agents seeking to minimise their cost functions will be pushed towards values with the least penalties rather than those with the least constraint violations. To overcome this, incremental penalties are reset to zero during the search, as follows:

- When a variable has a consistent value, all incremental penalties attached to each value in its domain are discarded. Here, we assume that penalties become redundant when an agent has a consistent value. This action also leaves room for maneuvering when either the variable suddenly becomes inconsistent or it is involved in a deadlock resolution process.

- Each agent also discards the penalties on its entire domain when it detects distortions to its cost function. We determine that a cost function is distorted when all the conditions in the rule, detailed in Figure 5.1, are satisfied. This second case is akin

to an aspiration move (as in Tabu search [34, 35]), where search memory is sometimes ignored. This is illustrated with the example in Figure 5.2. In the example, the cost function is distorted because the assignment $\langle x = g \rangle$ has the least sum of constraint violations and penalties. But, the assignment $\langle x = b \rangle$ now violates fewer constraints and as such the cost function is being distorted by the penalties. Therefore, resetting penalties this way allows agents to keep paths to better solutions open.

Distortion Rule:

Condition 1.1: The evaluation (h) of the current assignment is the least in a variable's domain.

Condition 1.2: There is another value in the domain, which has fewer constraint violations than the current assignment.

Figure 5.1: Detecting distortions to cost functions by incremental penalties.

As argued, resetting penalties allow agents to keep paths to potential solutions open; but doing it quite frequently also comes with a potential to cause the search to oscillate as it continuously removes barriers that prevent it from returning to earlier visited (and infeasible) regions of the search space. In Section 5.4.2, where we discuss penalty resets in detail, we show that the benefits of doing this are significant and they outweigh the possible risks.

$Dx = [w, r, b, g, k]$

$v(Dx) = [3, 5, 2, 3, 6]$
$p(Dx) = [2, 0, 4, 1, 0]$

therefore,
$h(Dx) = [5, 5, 6, 4, 6]$

and the current assignment, with the least $h(Dx)$ is $\langle x = g \rangle$.

Figure 5.2: An example of a distorted cost function

To tie both phases of the deadlock resolution strategy together, each agent maintains a no-good store in which deadlocks encountered are kept as no-goods. A no-good is an agent's *AgentView* comprising the assignments of all variables constrained with it at the time of the deadlock (Definition 3.6). No-goods are not treated as new constraints, to rule

out infeasible tuples, as used in forms of learning prevalent in backtracking algorithms (e.g. [33, 9]). No-goods are simply used as short-term memory to enable agents to determine what phase of resolution to use when they are stuck. As such, when a deadlock is encountered an agent checks its no-good store to find out if the deadlock state has been visited recently. If not so, the deadlock state is placed in the no-good store and the agent proceeds to initiate the first phase of deadlock resolution. And if the deadlock state is in the no-good store, the agent recognises that at least one previous attempt had been made at resolving the deadlock and can proceed with a longer term approach to resolving it i.e. it applies the second phase of deadlock resolution.

A fixed number of no-goods is held at any point in time by each agent, maintained on a First-In-First-Out basis. From experiments, which are discussed in Section 5.4.3, we found that the maximum number of no-goods held is not particularly critical to DisPeL's performance. This is because, as we also show in that section, that the majority of no-goods are only encountered once during the search. Therefore, we chose arbitrarily to fix the maximum number of no-goods held by each agent to 4 irrespective of the number of constraints attached to the agent or the kind of problem being solved.

5.2.3 Agent behaviour

Agents take turns to improve an initial random solution in DisPeL and the order in which their turns are taken is decided using the Distributed Agent Ordering scheme [43]. 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) i.e. parents are those neighbours that precede an agent in alphabetical order. This results in a static ordering that permits concurrent activity by unconnected agents.

During the search, each agent communicates with both sets of neighbours sending updates to them, as well as penalty messages to lower priority neighbours. Agents with higher priority neighbours get to take their turns after receiving messages from all of them. While agents without higher priority neighbours (i.e. Γ^+ is empty), only become active after receiving updates from all their lower priority neighbours; this prevents them from

continuously sending updates and allows for proper synchronisation of each iteration.

At initialisation, agents select random values for their variables and inform all neighbours of their current assignments. After which, the agents take turns to improve the solution executing the processes outlined in Algorithms 5.1, 5.2, 5.3, 5.4, and 5.5. When active, each agent selects the value with the least cost i.e. minimising equation (1). And, where there are two or more values with the same sum of constraint violations and penalties, an agent selects the leftmost[3] of these values. If, however, there is no value with a lower evaluation than the current assignment then the assignment is retained. After selecting a value, the agent sends an update to all its neighbours informing them of the new assignment.

Algorithm 5.1 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:    sendMessage(penaltyRequest)
22:  end while
23: until termination condition met
```

Deadlock resolution is initiated whenever an agent detects that it is at a quasi-local optimum. At this stage, the agent checks its no-good store to find out if the deadlock has been recently encountered. If the deadlock is new, it imposes the temporary penalty on its

[3]This is similar to the leftmost minimum rule in [44, 66].

Algorithm 5.2 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 5.3 procedure **respond_to_message()** Responding to a penalty message received from a higher priority agent.

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

Algorithm 5.4 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
```

Algorithm 5.5 procedure **sendMessage**(*penaltyRequest*)

1: send **message**$(id, value, null)$ to all neighbours in Γ^+
2: **if** $penaltyRequest = IncreaseIncPenalty$ **then**
3: send **message**$(id, value, penaltyRequest)$ to all neighbours in Γ^-
4: **else if** $penaltyRequest = ImposeTemporaryPenalty$ **then**
5: send **message**$(id, value, ImposeTemporaryPenalty)$ to neighbours in Γ^- violating constraints with $Self$
6: send **message**$(id, value, null)$ to neighbours in Γ^- not violating constraints with $Self$
7: **else**
8: send **message**$(id, value, null)$ to all neighbours in Γ^-
9: **end if**

current value (Algorithm 5.2, line 6) and selects a new value minimising (1) (Algorithm 5.2, line 13). In addition, its *AgentView* is placed in the no-good store (Algorithm 5.2, line 7) and at the same time the agent sends a message to neighbours in Γ^- that were violating constraints with it[4] to impose temporary penalties on their current assignments (Algorithm 5.5, lines 4-6).

However, if the deadlock had been previously encountered, the agent increases the incremental penalty attached to its current assignment by 1 (Algorithm 5.2, line 10) and goes on to select a value minimising its cost function. Furthermore, while informing its neighbours of its new value, it also requests that all neighbours in Γ^- increase the penalties attached to their current assignments (Algorithm 5.5, lines 2 and 3).

When an agent receives a request to impose or increase a penalty on its value it does so accordingly and selects the least cost value in its domain (Algorithm 5.3). Agents may at times receive multiple requests from two or more neighbours in Γ^+ simultaneously, and when this happens the requests are treated as one request if they are for the same type of penalty. For example, the agent will not increase the penalty on its current assignment more than once in a single iteration even if it receives several requests to do so. But, when the requests are simultaneously received for different penalties, agents will ignore the requests to impose a temporary penalty (Algorithm 5.4) - thereby prioritising learning over perturbation and allowing agents to focus on resolving the deadlocks in the order in which they were encountered.

[4]That is before its value may have been changed as a result of the temporary penalty.

To keep things stable and limit the amount of simultaneous deadlock resolution activity, an agent involved in deadlock resolution, i.e. one that has received a request to impose a penalty in the current iteration, is prevented from initiating any deadlock resolution action itself (Algorithm 5.1, lines 10-12).

5.2.4 An example run

The DisCSP in Figure 5.3 is used to illustrate the deadlock resolution process in DisPeL. In this example, there is a deadlock between agents **b** and **d**; neither agent has a value in its domain that reduces the number of constraints violated, as shown in $v(b)$ and $v(d)$ in the figure. Agents in the DisCSP will take their turns in the order **a**, then **b**, and then **c**, **d**, and **e** can take their turns simultaneously since there are no constraints between them; after which agents **f** and **g** take their turns one after the other. Since, agent **a** is consistent we assume it has already taken its turn where it does not change its assignment.

At this point, the agent **b** imposes a temporary penalty on its assignment causing it to change its value to $\langle b = 2 \rangle$. It also sends a request to agent **d**, which was violating a constraint with it, asking **d** to impose the temporary penalty on its current value too (Figure 5.4). In response, agent **d** imposes the temporary penalty on its value, and selects the least cost value in its domain $\langle d = 3 \rangle$ (Figure 5.5).

Other agents are unaffected by the changes, and therefore will keep their assignments in the current iteration. In the next iteration, agent **b** evaluates its state and finds that the values 3 and 4 both violate a single constraint each and it selects the first of those values $\langle x = 3 \rangle$. This, in turn, prompts agent **c** to change its assignment to $\langle c = 2 \rangle$, as it had suddenly become inconsistent (Figure 5.6). At the same time, the deadlock between agents **b** and **d** is now resolved.

If the perturbation failed to resolve the deadlock and, all agents still have their original assignments, agent **b** would have initiated the next phase of resolution by increasing the incremental penalty on its current value, and asking all its lower priority neighbours (**c**, **d**, **e**) to do the same.

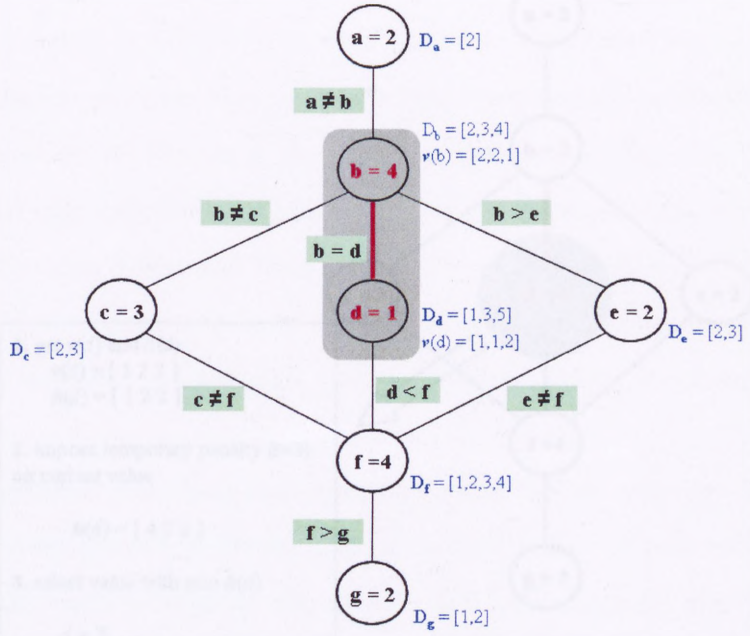

Figure 5.3: Illustration of deadlock resolution in DisPeL with the temporary penalty.

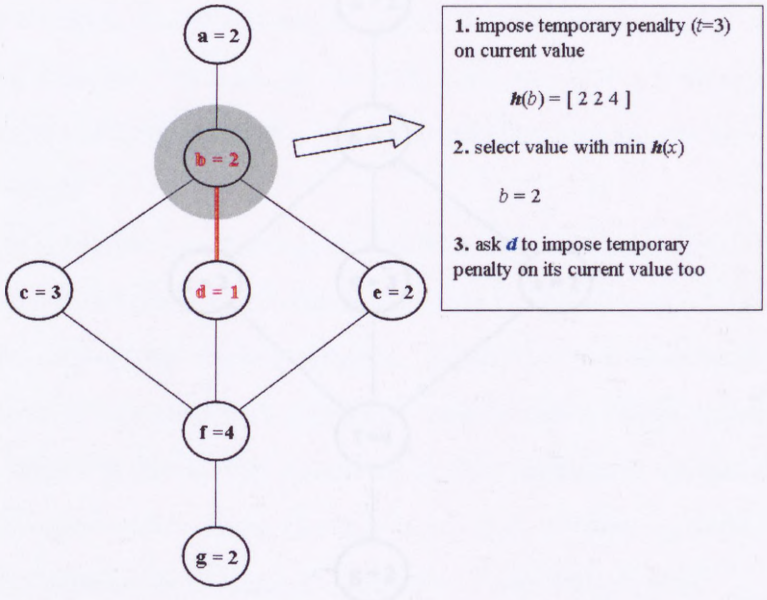

Figure 5.4: Example of deadlock resolution with the temporary penalty (step 2).

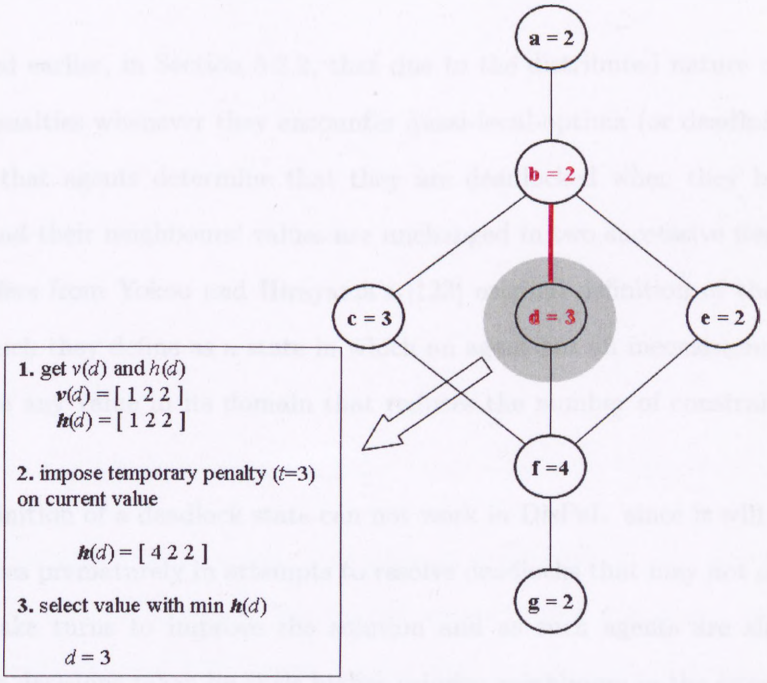

Figure 5.5: Example of deadlock resolution with the temporary penalty (step 3).

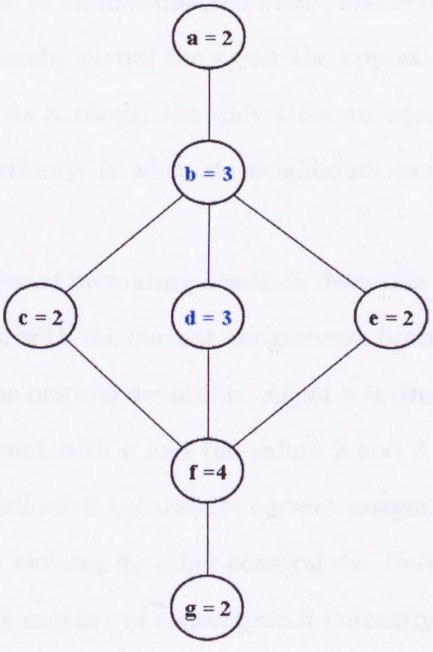

Figure 5.6: Example of deadlock resolution with the temporary penalty (step 4).

5.3 Deadlock detection in DisPeL

We mentioned earlier, in Section 5.2.2, that due to the distributed nature of the search, agents use penalties whenever they encounter quasi-local-optima (or deadlocks). We also pointed out that agents determine that they are deadlocked when they have inconsistent values and their neighbours' values are unchanged in two successive iterations. This approach differs from Yokoo and Hirayama's [123] original definition of the quasi-local-optimum, which they define as a state in which an agent has an inconsistent value and it does not have any value in its domain that reduces the number of constraints currently violated.

Their definition of a deadlock state can not work in DisPeL since it will cause agents to use penalties prematurely in attempts to resolve deadlocks that may not exist. Agents, in DisPeL, take turns to improve the solution and as such agents are simultaneously responding to decisions taken by their higher priority neighbours in the current iteration, and those taken by lower priority neighbours in the preceding iteration. Therefore, if an agent assumes it is deadlocked because it currently has no improvements there is a probability that in response to an unconnected event, one of its lower priority neighbours might change its value; thereby giving the agent the opportunity to try other values in the succeeding iteration. As a result, the only time an agent can determine that it is deadlocked, with some certainty, is when its neighbours assignments are unchanged in successive iterations.

We illustrate the problem of premature deadlock detection further with the DisCSP in Figure 5.7. In the example, with the current assignments both agent **a** and agent **c** are at quasi-local-optima given the original definition. Agent **a** is deadlocked because its domain value 1 violates the constraint with **c** and the values 2 and 3 violate the constraint with **d**. Similarly, agent **c** is deadlocked because its current assignment violates the constraint with **d** and its other value violates its other constraints. Neither agent has any value in its domain that reduces the number of constraints it currently violates.

Using DisPeL's approach, agents **a** and **c** will note their current states, retain their current assignments, and wait for the next iteration to confirm the deadlocks. However,

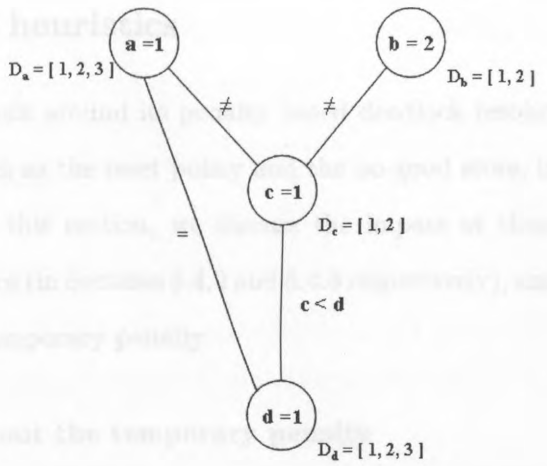

Figure 5.7: Examples of "premature" deadlocks in DisPeL's framework.

when it gets to agent **d**'s turn to act, it changes its assignment to $\langle d = 2 \rangle$ which opens an opportunity for agent **a** to resolve the conflict without using penalties. And, as a result the possible deadlock which both agents **a** and **c** were preparing to resolve no longer exists.

If either agent **a** or **c** had acted immediately, any penalties implemented could have influenced agent **d** such that it could have been prevented from making the change that dissolves the deadlock. The effects could possibly have pushed the search further away from nearby solutions. As a result, agents can miss opportunities to intensify the search of the surrounding regions of their current location in the search space.

In Chapter 7, this deadlock detection scheme is not used in the extension of DisPeL for agents with multiple local variables because of the peculiarities of deadlock detection in such problems i.e. deadlocks may be local or the interaction of inter and intra-agent constraints may increase the complexity of the deadlock detection process. We found that performance of that extension is not critically hampered by the absence of the deadlock detection mechanism, but we must stress that features of the problems studied may be a factor. Removal of deadlock detection in DisPeL may be of some benefit in highly structured problems which local search algorithms traditionally do not fare well. It should prevent agents from getting stuck in deep plateaus in the skewed cost landscapes of structured problems, therefore the removal of deadlock detection could possibly improve overall search efficiency.

5.4 Impact of heuristics

DisPeL is primarily built around its penalty based deadlock resolution strategies but its other components, such as the reset policy and the no-good store, influence its behaviour in different ways. In this section, we discuss the impact of these components on the algorithm's performance (in Sections 5.4.2 and 5.4.3 respectively), and also look empirically at the impact of the temporary penalty.

5.4.1 DisPeL without the temporary penalty

The temporary penalty is used in the first phase of deadlock resolution to perturb the solution state. The justification for its inclusion, in Section 4.4, is evidence showing that when used it has advantage of not causing as many new constraint violations in other parts of a problem not affected by a deadlock. Here, we study, empirically, the effect of the temporary penalty on DisPeL's performance.

DisPeL's main strategy for dealing with local optima is the landscape modification with penalties. This is a "long term" approach for resolving deadlocks and also for preventing their repeated occurrence. The temporary penalty is a perturbation scheme that also creates the effect of a rapid build-up of incremental penalties. In this guise, it aids DisPeL by speeding up resolution of some deadlocks that would otherwise require a build up of incremental penalties. It therefore, allows for quicker resumption of search intensification activity.

To examine the impact of the temporary penalty on DisPeL's behaviour, an experiment was conducted comparing DisPeL with two reduced versions of it: one where incremental penalties alone are used for deadlock resolution, and another where the temporary penalties are used exclusively. Performance was evaluated with two sets of randomly generated DisCSPs ($\langle n = 60, d = 10, p1 = 0.1, p2 = 0.5\rangle$ and $\langle n = 100, d = 10, p1 = 0.06, p2 = 0.5\rangle$), where we measured the percentages of problems solved within a maximum of $100n$ iterations. There were a thousand problems in each problem set, although we used only the first 100 problems (from one set) to evaluate the version relying exclusively on the

temporary penalty[5]. The summary, in Table 5.1, also includes the average and median number of iterations taken to find the solutions[6].

Heuristic	n	% of problems solved	average cost	median cost
Temporary penalty alone	60	4.0	215.5	172.0
Incremental penalty alone		82.7	801.5	451.0
DisPeL		87.6	1107.5	661.0
Incremental penalty alone	100	74.5	1613.9	905.0
DisPeL		88.6	1660.0	988.0

Table 5.1: Influence of temporary and incremental penalties on DisPeL's performance.

The results of the experiments show that with the temporary penalty alone, the algorithm rarely solved any problems. The reason for this is that, there is no strategy for detecting repeat visits to deadlock states and therefore the search is nearly always locked in an infinite oscillation between deadlock states. This can occur when perturbations continuously push the search back to points earlier in its trajectory, or when some perturbations conspire to cause the search to repeatedly jump back and forth between some deadlocks.

Combined with the incremental penalty in DisPeL, the contribution of the temporary penalty is a higher percentage of problems solved; compared with the version of the algorithm using incremental penalties alone. The gap between the algorithms is wider in the larger problems - where the percentage of unsolved problems by DisPeL was about half of that using incremental penalties alone. However, the costs do appear to be lower in the fewer problems solved with the incremental penalties alone. We argue that the temporary penalty improves the performance by allowing the algorithm resolve some "easy" deadlocks quickly, and thus improving chances of finding the solution within the time bounds by allowing the algorithm to use the incremental penalties only on the more difficult deadlocks. Furthermore, as we have shown in Section 4.4, when the temporary penalty is used the likelihood of causing previously satisfied constraints to become violated is lower.

[5]The version with the temporary penalty alone was not used for all problems in the set and it was not considered for the experiments with the second problem set because preliminary investigations had indicated that the version was not competitive.

[6]Unsolved problems are excluded from the statistics.

5.4.2 The penalty reset policy

Incremental penalties are reset when agents have consistent assignments and when they detect distortions to their cost functions. We argue that this is necessary because the penalties can dominate cost functions therefore forcing agents to seek values with the least penalties and at the same time blocking paths to solutions. Similar ideas of discarding search memory (in the form of weights or penalties) have been explored in the literature, especially in the work on centralised local search. For example, periodic penalty resets were proposed in [75], while regular [30] and probabilistic [57] weight decays have been shown to improve performance of weighted local search algorithms. The authors also argue that weights can block paths to solutions when retained. Tompkins and Hoos [111], in their study of the effects of weights on cost landscapes, conclude that weights often have large unintended effects on landscapes and that there must be mechanisms to undo such effects.

Both conditions for resetting penalties in DisPeL are new. As far as we are aware, there is no equivalent in the literature for resetting penalties when agents (or variables in centralised search) find consistent assignments. We argue that doing this gives agents room to maneuver when they suddenly become inconsistent or have to partake in a deadlock resolution with inconsistent neighbours. The results summarised in Table 5.2 are from experiments used to justify our reset policy, where we compared DisPeL with variants of it using different reset policies: not resetting penalties at all, resetting penalties only when consistent values are found, and resetting penalties only when cost functions are distorted. In addition, a version of DisPeL with continuous penalty resets is also included in the experiments.

Policy	% of problems solved	average cost	median cost
No resets	0	n/a	n/a
Reset only when consistent	87	1725	1207
Reset only when distorted	84	1597	1142
Combined resets (DisPeL)	93	1040	676
No penalty retention	48	1026	673

Table 5.2: Comparative evaluation of alternative reset policies in DisPeL on attempts to solve 100 randomly generated DisCSPs $\langle n = 60, d = 10, p1 = 0.1, p2 = 0.5 \rangle$.

First of all, the results in the table show that the impact of any form of penalty resets is significant in this framework. Penalties differ from constraint weights fundamentally in the way they affect cost functions. While one can look at weights as being 'woven into the fabric' of an underlying function, penalties are more like appendages. Hence, while solvers that rely on constraint weights can successfully solve problems without limiting the growth of weights, it appears that it is not the case with this penalty based strategy. It is clear that penalties do a very good job of blocking off paths to solutions if retained. As Table 5.2 shows, performance improves when penalties are discarded frequently; more problems are solved and the search costs are at least 40% lower.

The trend in Table 5.2 shows that performance of the algorithm improves as there are more opportunities to discard penalties (i.e. for the first four rows). This naturally leads to the question of what happens if a maximal reset policy is used i.e. incremental penalties are discarded as frequently as the temporary penalties. Results from a version of DisPeL with this policy show that the percentage of problems solved drops dramatically, and it suggests that the policies implemented in DisPeL allow the algorithm to retain penalties as long as they are useful and therefore properly learn about assignments associated with quasi-local-optima.

5.4.3 Impact of the number of no-goods held

No-goods are held by agents primarily to allow them to decide on what penalty to implement when deadlocks are encountered. But no-goods also serve as a form of memory where, in a way, they can help agents detect cycles when the search oscillates between a few deadlock states. That is if at least two no-goods are held. Such oscillations can occur when a perturbation at a deadlock pushes the search towards another deadlock, and a second perturbation pushes the search back to the first deadlock. If only one no-good is retained, agents cannot discover this oscillation and will waste a lot of effort roaming about in a small region of the search space. This can possibly prevent the algorithm from solving the problem.

No-goods are not taken as new constraints, and therefore, agents only hold a limited

number of them at any point in time. In this section, we examine what impact the number of no-goods agents are permitted to hold can have on DisPeL's performance. So far we have shown that other forms of search memory, i.e. the incremental penalties, improve DisPeL's performance if they are short lived, and we try to see if the same applies to no-goods. We carried out two sets of empirical experiments to flesh out any influences that the number of no-goods held can have on DisPeL's behaviour. To start with, we studied the frequency of repeat visits to deadlock states by DisPeL, if all no-goods are retained in memory with a view to establishing how often no-goods are referred to, on average, during the search and look for pointers to how much information needs to be retained. In the second set of experiments, we evaluate DisPeL's performance with varying limits on the maximum number of no-goods agents were allowed to hold at any point in time. In those experiments, we compared runs over several problems and used Run Length Distributions [55] with single instances to further scrutinise the behaviour.

In the first experiment, we ran DisPeL on several problems of different sizes, i.e. number of variables and domain sizes, to study how often deadlocks encountered are revisited in the course of a search. In these runs, agents were allowed to hold every deadlock they encountered. Results of this experiment, which are summarised in Table 5.3, show that at least 60% of the deadlock states were only encountered once during the search. And, at most 20% of deadlocks were revisited more than twice during the search. On the smaller problems, the percentage of repeat visits is higher. There appears to be a lot of explorative activity as the size of the search space grows and agents are not being repeatedly attracted to the same deadlocks. Hence, suggesting that there is no need to retain too much information, as only a handful of deadlocks turn out to be significant and require more attention for their resolution.

We test this assertion in the second experiment where we ran DisPeL on several problems with different limits to the number of no-goods agents were permitted to hold. We tried some arbitrary limits and also ran the algorithm with individual limits for agents - in multiples of the number of constraints (C) attached to their variables. Two problem sets were used for this experiment, each comprising 100 randomly generated DisCSPs. Table

Problem set	% 1 visit (σ)	% 2 visits (σ)	% > 2 visits
$\langle 40, 5, 0.15, 0.5\rangle$	61.2 (5.64)	19.0 (4.4)	19.8
$\langle 40, 10, 0.15, 0.5\rangle$	66.9 (15.29)	19.8 (14.7)	13.3
$\langle 60, 10, 0.1, 0.5\rangle$	68.0 (2.53)	19.2 (2.1)	12.9
$\langle 60, 5, 0.1, 0.5\rangle$	74.3 (12.86)	16.0 (11.0)	9.7
$\langle 75, 8, 0.08, 0.5\rangle$	74.9 (3.21)	14.7 (3.6)	10.4
$\langle 75, 15, 0.08, 0.5\rangle$	70.4 (0.91)	15.9 (0.5)	13.7
$\langle 80, 15, 0.08, 0.5\rangle$	74.2 (0.96)	15.5 (0.7)	10.3
$\langle 80, 8, 0.08, 0.5\rangle$	76.0 (2.96)	13.4 (2.5)	10.6

Table 5.3: Frequency of visits to deadlock states. Average (and standard deviation) from runs on 50 random DisCSPs in each set.

5.4 summarises the results of the experiment, showing the percentage of problems solved, the average and median search costs from the runs. In all the runs, the algorithm was started from the same initialisation to remove any random influences on the outcome.

n	$ngMax$	% solved	average cost	median cost
40	1	82	593.3	314.0
	2	90	627.6	338.5
	4	90	684.2	389.5
	8	90	650.3	406.5
	C	87	653.9	423.0
	2C	89	620.8	402.0
60	1	85	1124.1	677.0
	2	91	1268.5	760.0
	4	90	1176.9	646.0
	8	87	1014.8	580.0
	C	90	1039.7	647.5
	2C	91	1291.8	772.0

Table 5.4: DisPeL's performance on random DisCSPs ($\langle n = 40, d = 10, p1 = 0.15, p2 = 0.5\rangle$ and $\langle n = 60, d = 10, p1 = 0.1, p2 = 0.5\rangle$) with different limits ($ngMax$) on the number of no-goods agents hold.

As expected, the results in the table show that when only one no-good is held, the probability of finding solutions is lower. As we have argued earlier, holding one no-good can prevent agents from detecting oscillations between deadlock states and thus prevent the search from converging on a solution state. But the results show that beyond holding one no-good, the effect of the limit is unclear, even when it is tailored to the individual problem. There are no clear patterns in the statistics, and this suggests that the other components of DisPeL dominate this particular parameter.

As a way of confirming this, we used Run Length Distribution costs to study how the probability of finding solutions changes with the number of iterations. We tested the algorithm with limits of 2, 4, and $2C$ no-goods per agent on a random DisCSP instance $\langle n = 40, d = 10, p1 = 0.15, p2 = 0.5 \rangle$. The plot in Figure 5.8 shows and example of the typical distribution of costs from 200 attempts with each value for the parameter. The curves overlap each other, and the probability of finding a solution grows at similar rates for the different values.

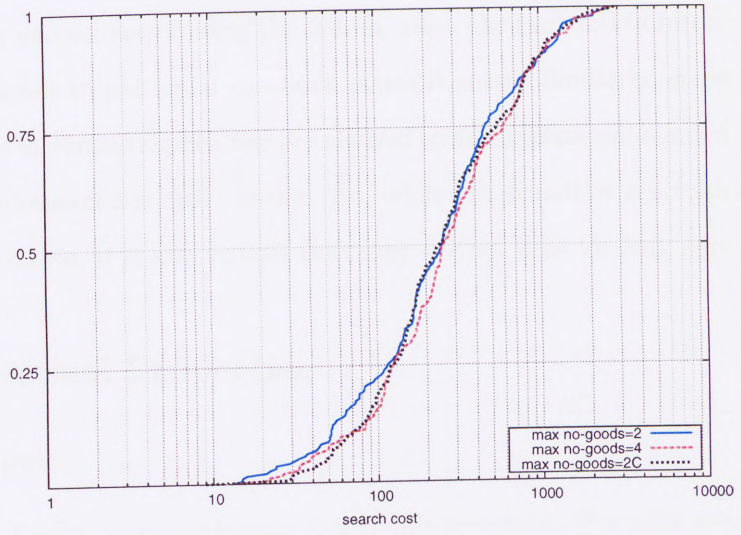

Figure 5.8: Run Length Distributions comparing performance of DisPeL when each agent holds a maximum of 2, 4, and $2C$ no-goods, plotted from 200 attempts on a random DisCSP.

In summary, we have shown that the number of no-goods held by agents is not particularly critical to DisPeL's behaviour. Except in the case where each agent holds just one no-good, the evaluations show that the effect of retaining more memory is unclear. Though a number of values are equally good, we arbitrarily chose to limit the maximum number of no-goods agents can hold to 4 - irrespective of the type of problems being solved or the problem size. We used this value in the rest of our empirical evaluations with DisPeL.

Summary

In the preceeding sections, DisPeL was stripped apart to study the impact of some of the prominent components of its strategy. Justifications for the inclusion of these components were outlined and backed up with empirical evidence. In summary, we have shown that although the temporary penalty (or perturbation phase) on its own is weak, it provides a massive boost to the performance of the algorithm when combined with the incremental penalty. We have also shown that agents do not necessarily need to retain so much memory about deadlocks encountered during the search, since the probability of the search being repeatedly attracted to particular deadlock states is small. Similarly, empirical evidence shows that when incremental penalties are allowed to accumulate unbounded, their impact on DisPeL's performance is quite severe. So, while the penalties are critical to helping DisPeL leave plateaus, it is also critical that they are not kept for long periods of time.

5.5 Theoretical Properties

5.5.1 Soundness

Theorem 5.1 *DisPeL is sound because it will only terminate iff a valid solution is found.*

Proof DisPeL will only report a solution when all agents stop and settle in a stable configuration. As long as there is a deadlock in the constraint network (there is at least one violated constraint), the highest priority agent involved in the deadlock will always take actions for its resolution. Therefore, the system will not settle on a stable configuration and, as a result, the algorithm will not terminate.

5.5.2 Completeness

DisPeL is incomplete because if a problem is unsolvable, agents have no way of detecting that fact. Search memory (i.e. penalties and no-goods) in DisPeL are ephemeral and therefore can not be used to cut out all infeasible areas of the search space. And as such, there is nothing theoretically preventing the search from repeatedly visiting previously

encountered local optima and running indefinitely. For the same reasons, DisPeL is not guaranteed to find a solution even if one exists.

5.5.3 Space complexity

DisPeL has a space complexity that is linear in the number of variables in the DisCSP being solved, and is bounded by $O(\sum_{i=1}^{|X|} D_i + (5 \times AgentView_i))$. The space used by each agent includes a matching penalty vector for its domain, a maximum of 4 no-goods held at any point in time, and the $AgentView$ from the previous iteration for deadlock resolution.

5.5.4 Privacy

In terms of privacy, agents in DisPeL preserve the same level of privacy as the prominent distributed iterative improvement algorithms. Agents reveal values that have the best evaluations in their domains and only one value is revealed at a time. Information "leaks" may occur if an agent's neighbour chooses to keep track of all values received from it over the course of the search. However, unless explicitly informed there is still some uncertainty, from the neighbour's perspective, about what extent of the agent's domain that has been revealed. Agents in DisPeL are not permitted to reveal information to one neighbour about their connections to other agents, nor are they permitted to inform neighbours about the values received from other agents.

Of course, this level of privacy is lower than that preserved in algorithms that use trusted servers (e.g. [125]) or secure encryption schemes (e.g. [104] and [84]). However, it is much higher than the level preserved in distributed backtracking algorithms with no-good learning; where the creation of no-goods can result in an agent informing one neighbour about constraints with another set of agents.

5.5.5 Termination detection

Termination detection in DisPeL is built around the fact that stability persists as soon as a solution is found. At a stable state, all agents will retain their current values and the solution remains rooted at a fixed position in the search space. Therefore to terminate

correctly, one needs to detect the fact that each agent is consistent and that the solution is unchanged in two successive iterations.

For experimentation, we took a pragmatic approach and assumed the existence of a System agent (as done in [40] and [105] for example), that initiates the search, handles message passing, and performs termination detection. Extending this approach for a distributed environment will require each agent to notify the System agent any time it has been consistent in two or more successive iterations. The System agent terminates the search if it has received such messages from all agents in a single iteration.

Alternatively, if the cost of regularly communicating with the System agent is high, the termination detection mechanism introduced in [123] can also be used in DisPeL. This mechanism is intertwined with the search algorithm, so that termination detection is not run as separate process and it does not increase the number of messages exchanged. We found that this mechanism also works well with DisPeL. Details of the mechanism are outlined in the aforementioned paper, as well as the proof of its correctness.

5.6 Empirical Evaluation

We carried out extensive empirical evaluations of DisPeL's performance with several types of problems including distributed graph colouring, random DisCSPs, and the car sequencing problem. Performance was evaluated with two metrics: (1) the percentage of problems solved within stated time limits, and (2) the cost of finding solutions, where, the cost metric is the number of iterations required to find solutions. This is used because in this particular case it is representative of other evaluation metrics, such as message counts and consistency checks, commonly used in the community. DisPeL is a synchronous algorithm in which all agents do consistency checks and communicate with their neighbours in each iteration. Therefore, the number of consistency checks and messages sent can be directly inferred from the number of cycles executed. Clock time is ignored as an evaluation metric because it is too implementation dependent [3]; and because all the experiments in this work were carried out in a simulation of a distributed system on a single machine. Therefore, the clock time from such simulations will not take into account the real costs

of distributed computation, such as message count, which are critical and can easily be inferred from the iteration count.

DisPeL is compared to the Distributed Breakout Algorithm (DBA) [123], which is the only other distributed iterative improvement algorithm that deals with local optima by modifying cost landscapes. In DBA, weights are attached to constraints and agents try to minimise the weighted sum of constraint violations. Weights are used to modify the landscape, by increasing those attached to violated constraints whenever agents encounter quasi-local-optima. Therefore, the comparison of DisPeL and DBA allows us to evaluate the effectiveness of the different landscape modification strategies, as well as to test the claims made in Section 4.3.

All agents in DBA act concurrently, each simultaneously looking for possible improvements (in an *improve* iteration) and exchanging current assignments (in an *ok?* iteration). While exchanging possible improvements, only those agents with the highest improvements in their neighbourhoods are allowed to change their values (ties are broken in favour of agents with the lowest lexicographic IDs). This prevents agents connected by constraints from changing values simultaneously and it allows the search to follow a steepest descent path.

Prior to the evaluation reported here, we verified our implementation of DBA by testing it on graphs generated with the same methods specified in [123] and we achieved results matching those reported. DBA and DisPeL were run on several DisCSPs where each variable was assigned to an agent and each agent represented just one variable. For the following experiments, we count each of DBA's cycles (the *improve* and *ok?* cycles) as a separate iteration to compare its search costs with DisPeL. Therefore, to give agents in DBA the same number of opportunities to improve a solution, the maximum number of iterations in each run for DBA is twice that used for DisPeL.

5.6.1 Distributed Graph Colouring

Keeping with tradition, our first set of experiments were conducted using random distributed graph colouring problems. These were all solvable instances created with the

method specified in [27]. For this class of problems, we were specifically interested in how performance changes with respect to constraint density (degree) on a fixed problem size. Therefore, we used the well studied 3-colour 100-node graphs for which complexity peaks are well established [18, 51]. For each constraint density considered, 100 random instances were generated, hence a total of 1,100. DisPeL and DBA were both run on these graphs and were limited to 10,000 and 20,000 iterations respectively. Results showing the percentage of problems solved, median search costs (i.e. number of iterations used), and average search costs are plotted in Figures 5.9, 5.10, and 5.11 respectively.

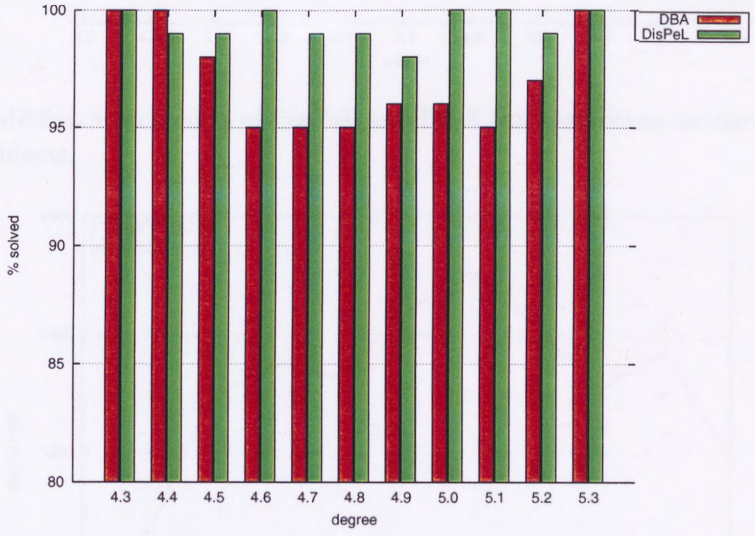

Figure 5.9: Percentage of distributed graph colouring problems $\langle n = 100, k = 3 \rangle$ solved by DisPeL and DBA.

Results show that on the percentage of problems solved, both algorithms have the "easy-hard-easy" profile which the problem class is known to exhibit. However, within the time limits, DisPeL solved more problems than DBA in all but one point. In the region of hard problems (i.e. $4.5 \geq \textit{degree} \leq 5.3$), DBA solved significantly fewer problems. In the profile of search costs plotted in Figures 5.10 and 5.11, DBA still exhibits that "easy-hard-easy" pattern, with search costs peaking in the middle where the hardest problems are. Although, DisPeL's search costs have a similar pattern these are almost always an order of magnitude lower than DBA's costs. And, the difference in search costs between "easy" and "hard" problems for DisPeL is not as pronounced.

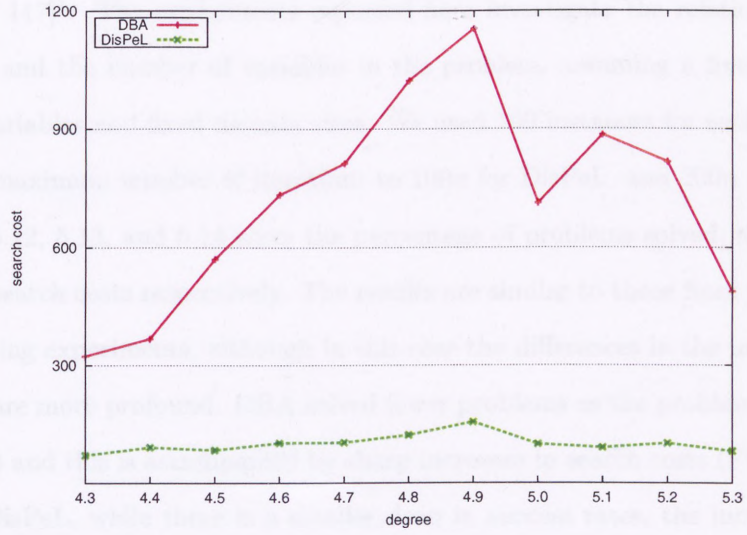

Figure 5.10: Median search costs of DisPeL and DBA from attempts on distributed graph colouring problems.

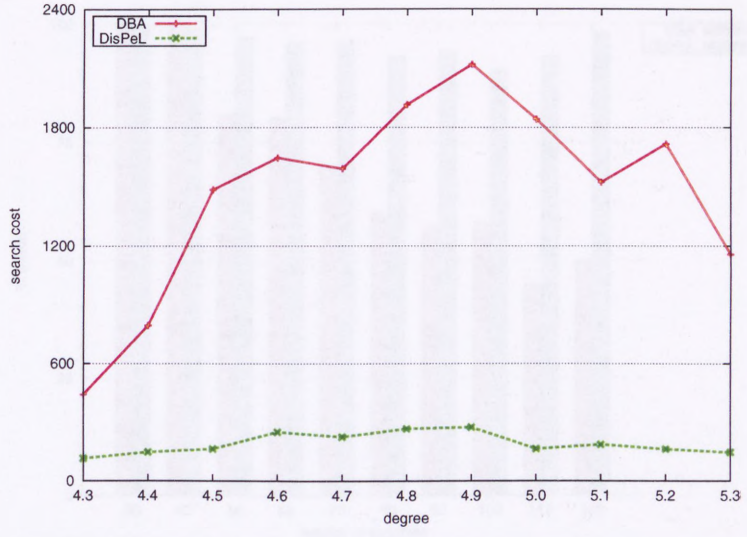

Figure 5.11: Average search costs of DisPeL and DBA from attempts on distributed graph colouring problems.

5.6.2 Random Distributed Constraint Satisfaction Problems

We conducted experiments on randomly generated distributed constraint satisfaction problems to study the behaviour of both algorithms on problems with non-binary constraints. These solvable instances were generated using the standard Model-B [85], but modified with preferential attachment of constraints to variables so that they resemble real-life

problems [6, 117][7]. The experiments reported here investigate the relationship between search costs and the number of variables in the problem, assuming a fixed ratio of constraints to variables and fixed domain sizes. We used 100 instances for each problem size, and set the maximum number of iterations to $100n$ for DisPeL and $200n$ for DBA.

Figures 5.12, 5.13, and 5.14 show the percentage of problems solved, the median and the average search costs respectively. The results are similar to those from the distributed graph colouring experiments, although in this case the differences in the number of problems solved are more profound. DBA solved fewer problems as the problem size increased (Figure 5.12) and this is accompanied by sharp increases in search costs (Figures 5.13 and 5.14). For DisPeL, while there is a smaller drop in success rates, the increase in search costs in relation to the problem size is nearly linear. Furthermore, search costs for DisPeL were significantly lower.

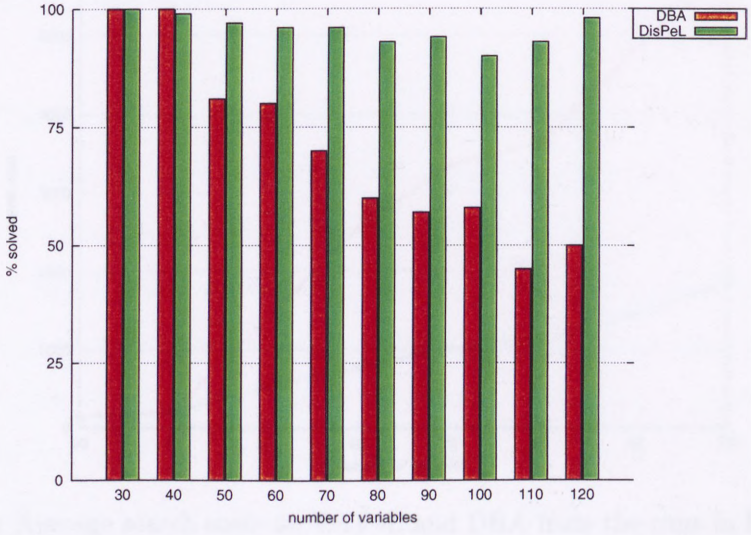

Figure 5.12: Percentage of problems solved by DisPeL and DBA from runs on problems with 3-ary constraints $\langle n \textit{ variables}, 2n \textit{ constraints}, d = 10, p2 = 0.55 \rangle$.

[7] In [6], it was shown that the distribution of links in many real life networks, as diverse as cell metabolic networks and the world wide web, follow a power law where the majority of nodes have a few connections to them and a small number of nodes have a high number of connections. Walsh [117] found similar patterns in his study of real life Constraint Satisfaction Problems and proposed a modified power law model for generating random realistic problems.

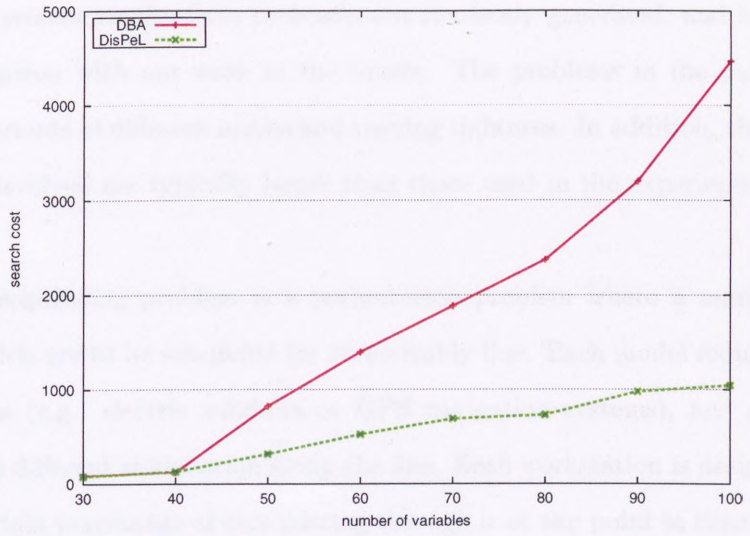

Figure 5.13: Median search costs for DisPeL and DBA from the runs in Figure 5.12.

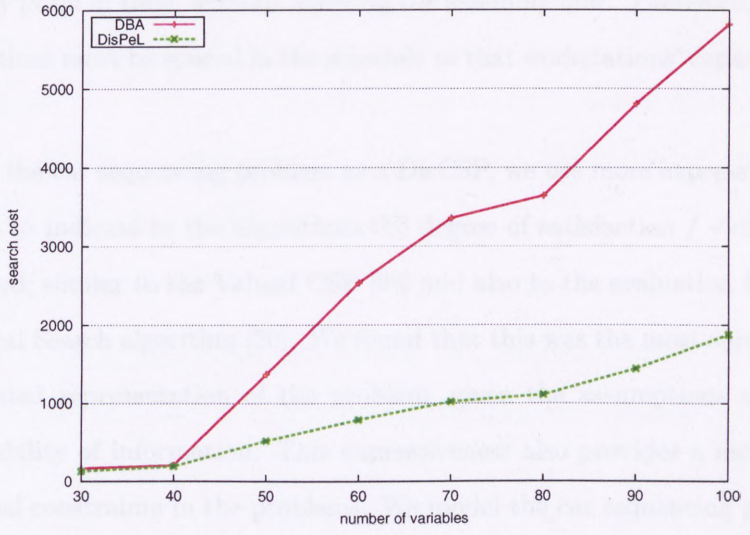

Figure 5.14: Average search costs for DisPeL and DBA from the runs in Figure 5.12

5.6.3 Car sequencing problems

Finally both algorithms were evaluated with Car Sequencing Problems [87]. Car sequencing is not one of the traditional test beds used for evaluating distributed algorithms, but it is used here because the car sequencing dataset[8] is one of the few publicly available problem sets where instances are entirely made up of non-binary constraints. Besides, it

[8]From the CSPLib at http://www.csplib.org

allows us to present results from problems not randomly generated, and leaves room for direct comparison with our work in the future. The problems in the dataset typically contain constraints of different arities and varying tightness. In addition, the domain sizes of variables involved are typically larger than those used in the experiments reported so far.

The car sequencing problem is a permutation problem where a number of cars of different models are to be scheduled for an assembly line. Each model requires a different set of options (e.g. electric windows or GPS navigation systems), and each option is installed by a different workstation along the line. Each workstation is designed to handle at most a certain percentage of cars passing through it at any point in time. For example, a workstation to install electric windows has the capacity to handle 3 out of every 5 cars on the line at any point in time, without affecting the assembly flow. Therefore, cars requiring particular options must be spaced in the schedule so that workstations' capacities are never exceeded.

To model the car sequencing problem as a DisCSP, we use more expressive constraints that allow us to indicate to the algorithms the degree of satisfaction / violation for each tuple evaluated; similar to the Valued CSP [98] and also to the evaluation function in the Adaptive Local Search algorithm [20]. We found that this was the most suitable approach for a distributed representation of the problem, given the assumptions of privacy and limited availability of information. This expressiveness also provides a means of dealing with the global constraints in the problems. We model the car sequencing problem in the DisCSP framework as a tuple $S = (M, W, X, A, D, C)$, where:

- $S = \langle s_1, s_2, ..., s_n \rangle$, is a the schedule (or solution) where each s_i is the slot in the ith position of the schedule.
- $M = \langle m_1, m_2, ..., m_k \rangle$ is the number of cars of k models to be assembled.
- $W = \langle w_1, w_2, ..., w_p \rangle$ is the set of options available and the respective workstations for installing each option.
- $O = \langle O_1, O_2, ..., O_k \rangle$, is the set of options to be installed on the cars of each model.

- $X = \langle x_1, x_2, ..., x_n \rangle$ the variables in the DisCSP, referring to the slots in the schedule therefore $x_i = s_i$.
- $A = \langle a_1, a_2, ..., a_n \rangle$ the agents representing variables in the problem, each variable belongs to only one agent and each agent represents only variable.
- $D = \langle D_1, ..., D_n \rangle$ the domain of each variable, $|D_i| = k$ where each value in D_i refers to a car model in M i.e. there is a one to one mapping between each D_i and M.
- $C = \langle Cap_1, ..., Cap_p; MC_1, ..., MC_k \rangle$ is the set of constraints comprising two types of constraints: (1) workstation capacity constraints (Cap) and (2) enumeration constraints (MC).

Each agent in the DisCSP is assigned a variable (or slot in the schedule) and is responsible for selecting the model of the car to be placed in its slot. Therefore, a solution is found when agents have built a schedule containing the right number of cars of each model and satisfy the capacity constraints of all workstations. The two types of constraints in our model of the problem are defined as follows:

Definition 5.1 (Capacity constraint $Cap_i \in C$). *A workstation capacity constraint $Cap(o_i, p, q)$ specifies that a maximum of p out of every q consecutive cars scheduled in S can require option i.*

Definition 5.2 (Enumeration constraint $MC_i \in C$). *An enumeration constraint is created for each model of car to be produced. Each enumeration constraint (MC_i) is a global constraint that is satisfied if the exact number of cars required for the ith model have been selected by agents.*

The scope of a capacity constraint ($Cap(o_i, p, q)$) is over every set of q consecutive slots in the schedule, such that for each option each $agent_j$ holds a capacity constraint for each sub-set of consecutive slots on the schedule including itself from $(j - q) + 1$ to $(j + q) - 1$. The example in Figure 5.15 is used to illustrate this.

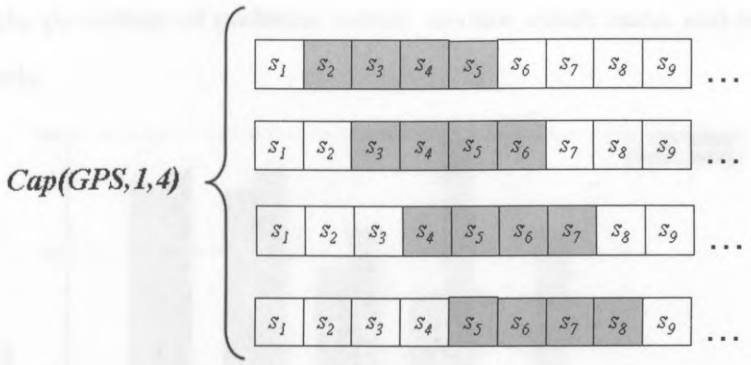

Figure 5.15: A illustration of scope of capacity constraints in the distributed car sequencing.

In Figure 5.15 we assume that the workstation installing GPS systems has a capacity to deal with at most one car out of every four consecutive cars in the schedule. Agent 5, in the example, keeps four copies of this constraint (shaded slots) one for each set of four consecutive cars including itself and evaluates each constraint separately. Agents evaluate each copy of the capacity constraint on its own so that the capacity constraints have a more expressive impact on the cost landscapes and can help to reduce the size and number of plateaus in that landscape.

Each agent keeps a copy of each global enumeration constraint and evaluates it with the choices made by other agents. Keeping with the expressiveness theme, the constraint evaluation returns a zero if the constraint is satisfied or the difference between the required number of cars and the number of cars of that model currently scheduled. Where a negative results indicates to agents that more of them need to consider selecting a particular model.

Solvable instances from the CSPLib dataset were used for the experiments. These are made up of 50 instances, grouped into 5 sets of 10 instances for each of the different workstation capacity rates (or constraint tightness) which range from 70% to 90%. In each of these instances there are 200 cars to schedule, 5 workstations, and 17 to 30 configurations of options (or models) to be considered. To generate enough data for analysis, we made 5 attempts on each problem instance starting off with a different random initialisation in each run. The maximum number of iterations was set to 5,000 and 10,000 for DisPeL and DBA respectively. Results of these experiments are presented in Figures 5.16, 5.17, and

5.18 showing the percentage of problems solved, median search costs, and average search costs respectively.

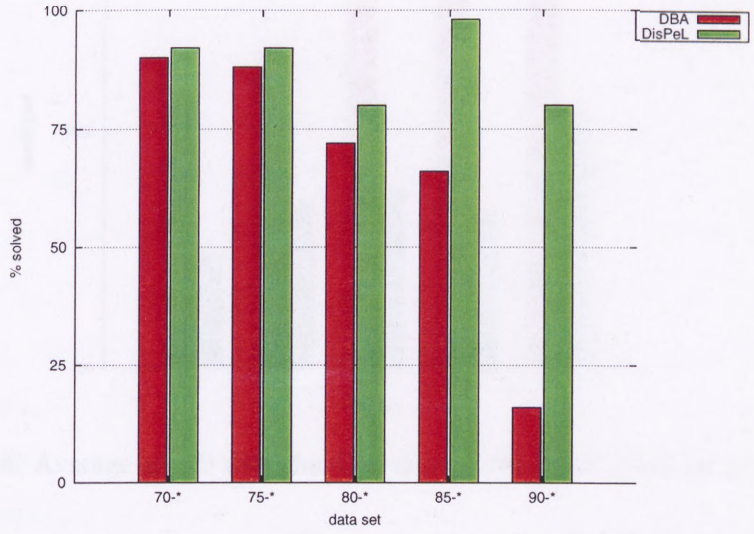

Figure 5.16: Percentage of distributed car sequencing problems solved by DisPeL and DBA.

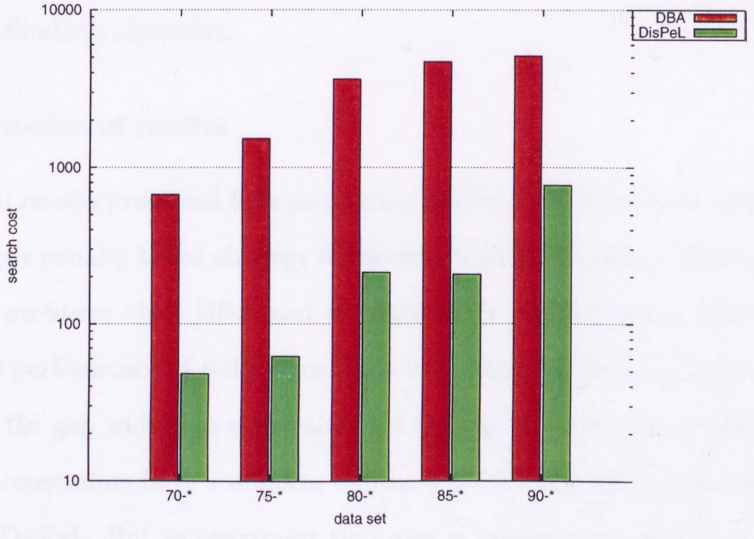

Figure 5.17: Median search costs for DisPeL and DBA from runs on in Figure 5.16.

The results are consistent with those reported in the previous experiments in Sections 5.6.1 and 5.6.2, where DisPeL consistently solved more problems and required fewer iterations. The results also show how DBA's performance deteriorates as the capacity constraints tighten. However, DBA is also handicapped by the problem structure. As

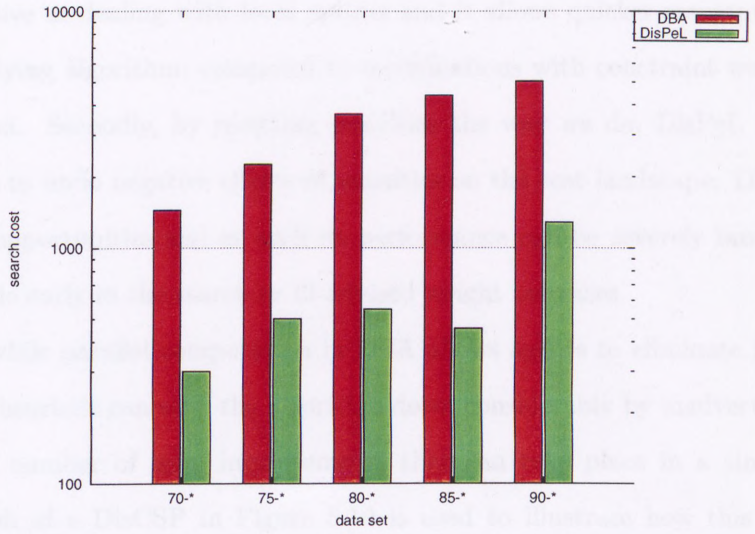

Figure 5.18: Average search costs for DisPeL and DBA from runs on in Figure 5.16.

the constraint graphs are complete, DBA's coordination heuristic only allows one variable to change its value every two iterations i.e. the *ok?* and *improve* iterations. There are no concurrent changes, thus DBA becomes the equivalent of a standard centralised local search / hill-climbing algorithm.

5.6.4 Discussion of results

The empirical results presented here comparing DisPeL and DBA have demonstrated the strengths of our penalty based strategy for dealing with local optima. DisPeL consistently solved more problems than DBA and it consistently required fewer cycles. While the differences in performance of both algorithms vary from one problem class to the next, it appears that the gap widens as constraints get tighter. For example, in distributed graph colouring all constraints have a uniform tightness of 30% and DBA solves nearly as many problems as DisPeL. But as constraint tightness is increased to 50% in the experiments with random DisCSPs, DBA's performance degrades considerably. This is even more evident in the results of the experiments on the car sequencing problems which include constraints with higher ratios of forbidden tuples.

We give three reasons to explain DisPeL's performance advantage over DBA. First of all, as we demonstrated in Section 4.2, landscape modification with domain penalties

is more effective at dealing with local optima and it allows quicker resumption of search by the underlying algorithm; compared to modifications with constraint weights - which DBA relies on. Secondly, by resetting penalties the way we do, DisPeL regularly has opportunities to undo negative effects of penalties on the cost landscape. DBA, however, has no such opportunities and as such its performance can be severely hindered by bad decisions made early in the search or ill-advised weight increases.

Thirdly, while parallel computation in DBA allows agents to eliminate idle time, the coordination heuristic can slow the algorithm down considerably by inadvertently cutting down on the number of legal improvements that can take place in a single iteration. The sub-graph of a DisCSP in Figure 5.19 is used to illustrate how this can happen. We assume that each agent in the illustration has computed the same possible improve given the current state of the solution, and that each of them has the best improvements amongst their other neighbours. Using DBA's coordination heuristic, agent **a**'s change is given priority over that of agent **b** and at the same time agent **b** has a priority over agent **c**. Therefore, in the current iteration only agent **a**'s value is allowed to change; even though agent **c** can also change its value without causing causing any oscillations as agent **b**'s value is fixed.

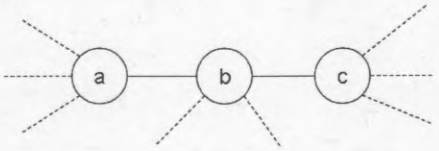

Figure 5.19: Preventing simultaneous changes in DBA - an illustration.

In Figure 5.20, we look at the resulting effect this on a larger scale where we show the number of agents changing values in every two iterations in DBA from a sample run on a distributed graph colouring instance $\langle n = 100, k = 3, degree = 4.7 \rangle$, plotted along side the same count from a sample run of DisPeL. The number of consistent agents in the corresponding iterations for both algorithms are plotted in Figure 5.21.

There is a lot more activity in each iteration of DisPeL than DBA (in terms of agents changing values) especially in the first few iterations and a high percentage of agents

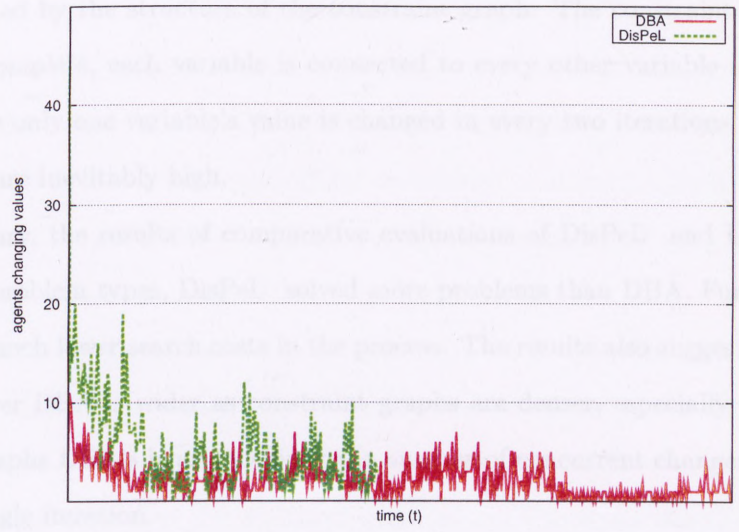

Figure 5.20: Number of agents changing values in each iteration from sample runs of DBA and DisPeL.

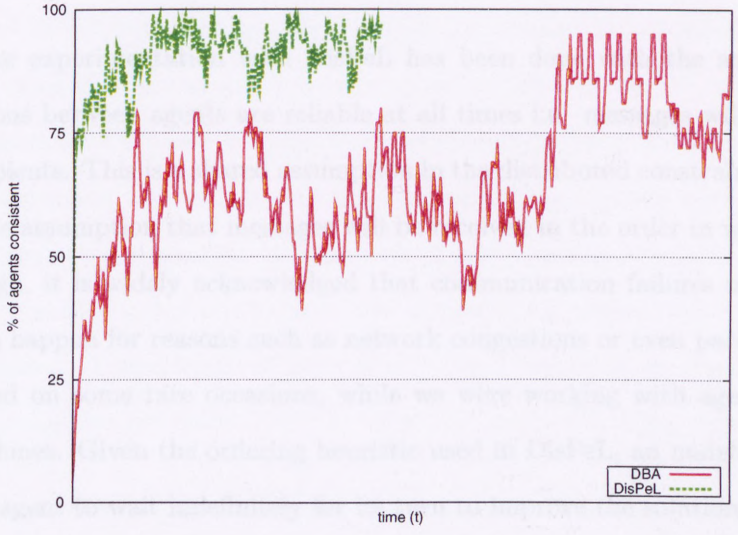

Figure 5.21: Number of consistent agents in each iteration from sample runs of DBA and DisPeL.

quickly becoming consistent in DisPeL. Therefore, it appears that only 'critical' deadlocks remain unresolved. For DBA, few agents get to change values in each iteration and therefore deadlocks tend to linger during the search. As a result, the number of consistent agents increases at a much slower pace than in DisPeL. The obvious implication is that DBA will tend to require considerably more time than DisPeL to solve problems.

The experiments with the car sequencing problems vividly illustrate how DBA is ad-

versely affected by the structure of the constraint graph. The constraint graph for each instance is complete, each variable is connected to every other variable in the problem, and therefore only one variable's value is changed in every two iterations. Consequently, search costs are inevitably high.

In summary, the results of comparative evaluations of DisPeL and DBA show that on different problem types, DisPeL solved more problems than DBA. Furthermore, DisPeL incurs much lower search costs in the process. The results also suggest that DisPeL's advantage over DBA is wider as constraint graphs are denser, especially since in highly connected graphs DBA's heuristics limit the number of concurrent changes that can take place in a single iteration.

5.7 Coping with unreliable communications

So far, all the experimentation with DisPeL has been done with the assumption that communications between agents are reliable at all times i.e. messages will always get to intended recipients. This is a shared assumption in the distributed constraints community, along with the assumption that messages will be received in the order in which they were sent. Although, it is widely acknowledged that communication failures may occur, and that they can happen for reasons such as network congestions or even packet corruption. This happened on some rare occasions, while we were working with agents located on separate machines. Given the ordering heuristic used in DisPeL, an undelivered message can cause an agent to wait indefinitely for its turn to improve the solution, and this wait can cascade through the network and effectively stop the search. We remedied this with a modification that allowed agents to resume activity if messages have not been received after a reasonable amount of time. The agents were allowed to assume that the neighbour's (i.e. the sender) value is unchanged and proceed with the search.

Channel reliability is a major issue in distributed computation in general. Messages may be lost, duplicated, or due to traffic on different routers they may arrive in a different order from which they were sent. The resulting impact on the processes depending on these messages can be severe. Message tagging with sequence numbers and acknowledgement of

delivery are common ways of dealing with unreliable channels [91]. But even this approach is not immune from the aforementioned problems - acknowledgement messages can also be lost in transmission. Nevertheless, taking this approach to guarantee even minimum levels of reliability may be too expensive for distributed constraint solving. It can result in an exponential increase in the number of messages exchanged, and when weighed against its benefits, the additional measures may not be justifiable.

Studies on the effects of channel unreliability in distributed constraint solving suggest that DisCSP algorithms can sometimes benefit from unreliability. In work presented in [26], it was shown that the element of randomness in communication delays can improve performance and robustness of distributed backtracking algorithms like the Asynchronous Backtracking and Asynchronous Weak Commitment Search, as well as reduce overall network load. Similar work in [78], where the underlying algorithm was allowed to select which messages must be reliably delivered, showed that the occasional lost message reduces the amount of work agents do and in some cases speed up the algorithm.

In this section, we studied the effects of lost messages on DisPeL's performance to test its robustness and also with a view to determine if there is any need to incorporate additional measures to guarantee certain levels of performance. We look at the effect on the algorithm's ability to solve problems, and the resulting search costs, if lost messages are simply ignored. Asides from the issue of ordering / coordination (mentioned earlier), message losses can also influence DisPeL's behaviour in the following ways:

- The obvious case of agents not receiving updates from neighbours and, as such, making decisions with outdated information.
- The other obvious case of an agent involved in a deadlock not receiving a message to implement a penalty on its value.
- The false negative of an agent assuming it is at a quasi-local-optimum even though one of its neighbours' value has changed. And the agent subsequently initiates the deadlock resolution process, therefore disrupting search with spurious penalty requests.

We study the effect of channel unreliability in three scenarios[9]: (1) where any individual message can be lost, (2) where only messages sent from agents that have either changed their values or made penalty requests are from time to time lost, and (3) where occasionally, all messages sent by an agent in one iteration are not delivered. To perform the experiments, we modified DisPeL slightly by removing the ordering / coordination heuristic so that all agents sequentially take turns to be active.

For the first scenario, we devised an experiment to evaluate the impact of undelivered messages where unreliability was simulated by randomly deciding with a probability lp if a message sent from an agent to another is lost. RLD analysis was used to study the direct impact of the lost messages, abstracting out the effects of random initialisations and problem structures. This was done using a critically difficult distributed graph colouring instance $\langle n = 100, k = 3, degree = 4.7 \rangle$, with which 500 attempts were made with different values of lp. In each attempt, the search was initialised from the same random position and a limit of 10,000 iterations was imposed. In Figures 5.22 and 5.22, we show plots for runs with loss probabilities $[0.05, 0.1, 0.15]$ and $[0.05, 0.2, 0.4]$ respectively; and a summary of the results of the full experiment appears in Table 5.5.

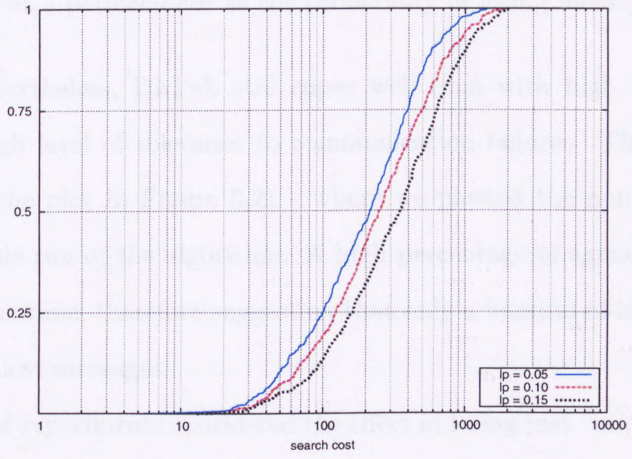

Figure 5.22: Distribution of DisPeL's search costs with message loss probabilities (0.05, 0.1, 0.15).

As expected, the plots show that search costs increase steadily with the number of

[9]We still assume that messages are received in the order in which they were sent and that there are no transmission delays.

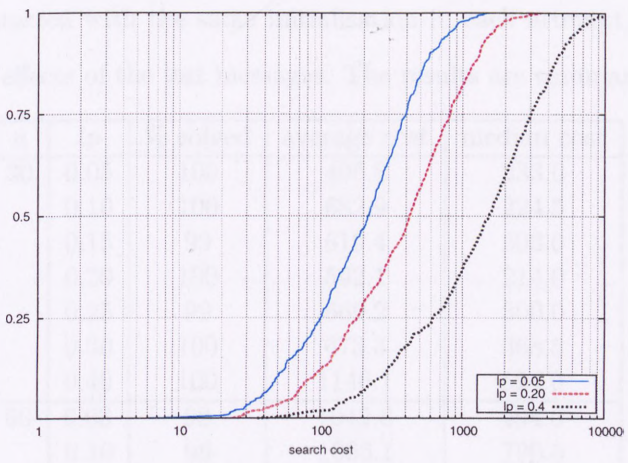

Figure 5.23: Distribution of DisPeL's search costs with message loss probabilities (0.05, 0.2, 0.4).

lp	% solved	average cost	median cost
0.05	100	284.2	215.5
0.10	100	363.5	253.5
0.15	100	459.1	350.5
0.20	100	579.7	422.0
0.25	100	759.1	577.0
0.30	100	1086.7	766.5
0.40	99	2070.3	1525.5

Table 5.5: DisPeL's performance as the probability of losing messages increases.

lost messages. Nevertheless, DisPeL still copes well even with high loss probabilities; demonstrating a high level of tolerance to communication failures. This robustness can be explained with the plot in Figure 5.21 - where we plotted the number of consistent agents from a sample run of the algorithm. A high percentage of agents in the run were consistent at any one time, therefore suggesting that only a handful of inconsistent agents may be affected by lost messages.

The second set of experiments considered the effect of losing just "important" messages i.e. only those messages where either an agent has changed its value or it is sending a request for its neighbours to implement a penalty. A case where as more messages are lost, agents will be making more decisions with inaccurate information about their neighbours assignments. We used two sets of random DisCSPs ($\langle n = 30, d = 10, p1 = 0.2, p2 = 0.5\rangle$ and $\langle n = 60, d = 10, p1 = 0.1, p2 = 0.5\rangle$) in this experiment, with 100 problems in each

set. DisPeL was started with the same initialisation in each attempt, so that we could focus solely on the effects of the lost messages. The results are summarised in Table 5.6.

n	lp	% solved	average cost	median cost
30	0.05	100	495.8	233.0
	0.10	100	682.9	224.5
	0.15	99	619.4	303.0
	0.20	100	592.1	214.0
	0.25	99	669.2	300.0
	0.30	100	673.3	368.5
	0.40	100	1146.1	524.0
60	0.05	98	1042.8	494.0
	0.10	96	1066.1	790.5
	0.15	97	1102.8	650.5
	0.20	95	1154.9	616.0
	0.25	97	1377.8	660.0
	0.30	97	1501.4	807.0
	0.40	95	1224.6	751.0

Table 5.6: DisPeL's performance as the probability of losing important messages increase.

A similar tolerance to communication failures is shown even when all lost messages are critical to agents' decision making. A high percentage of problems are still solved even with loss probabilities as high as 40%. The results for the runs on the smaller problems show gradual increases in search costs, with two sudden jumps, one between $lp = 0.05$ and $lp = 0.1$, and the other between $lp = 0.3$ and $lp = 0.4$; where the average search costs increase dramatically. But the trend is broken briefly at $lp = 0.2$. Results for runs on the larger problems also show the same gradual increase in search costs, although the algorithm nearly always solved the same percentage of problems.

Finally, we investigate the effects of occasionally "cutting off" some agents on DisPeL's performance. In this experiment, channel unreliability is simulated by dropping all messages sent by an agent (in an iteration) with the probability lp. And again, focusing on those messages that contain new information for the recipients. Similar problems to those used in the previous experiment (i.e. those in Table 5.6) were used and the results are summarised in Table 5.7.

The results show that performance is not steadily decaying, especially in the runs on the smaller problems. Rather, the effects of lost messages are almost random. But, the

n	lp	% solved	average cost	median cost
30	0.05	95	653.0	256.0
	0.10	97	678.8	321.0
	0.15	96	790.3	303.5
	0.20	96	639.4	255.0
	0.25	95	532.0	187.0
	0.30	97	723.9	371.0
	0.40	96	789.4	455.5
60	0.05	91	1880.4	724.0
	0.10	90	1917.3	903.0
	0.15	96	1515.1	926.0
	0.20	93	1703.2	1012.0
	0.25	93	1648.2	947.5
	0.30	95	1720.5	927.5
	0.40	93	1841.0	873.5

Table 5.7: DisPeL's performance as the probability of cutting of agents with important messages increase.

random loss of messages appears to help DisPeL through the introduction of randomisation that enables it avoid propagating the effects of bad decisions made by agents. As such, a high percentage of problems are still solved even with a 40% loss rate.

Overall the results show that DisPeL has a high tolerance for unreliable communications, and suggest that perhaps for just the worst case where message delivery is highly uncertain, it may not be necessary to incorporate additional measures such as acknowledgement messages to guarantee reasonable levels of reliability.

5.8 Chapter Summary

We described the Distributed Penalty Driven Search algorithm in this chapter. DisPeL is an iterative improvement algorithm that deals with local optima by perturbing the search and modifying the cost landscape with two types of penalties on domain values. It also includes additional heuristics, such as agents maintaining no-good stores and a policy of discarding penalties. The impact of each component of DisPeL's strategy was discussed in detail, and as a result we were able to show how DisPeL's performance is built on a synergy resulting from the combination of its components.

Results of empirical evaluations, and a comparison with the Distributed Breakout Algo-

rithm, were also presented in the chapter. The algorithms were evaluated with distributed graph colouring problems, random DisCSPs, and instances of the car sequencing problem. In all cases, results showed that DisPeL consistently solved more problems than DBA and required fewer iterations to do so. We argued that, in addition to the effects of constraint weights on the landscape, DBA was also handicapped by its coordination heuristic which limited the progress towards solutions. Finally, DisPeL's tolerance to unreliable communications was discussed. Results from empirical tests showed that with message failure rates as high as 40%, DisPeL was still able to find solutions in all attempts on a problem instance. Those experiments introduced some form of randomisation to DisPeL i.e. with the random loss of messages, in Chapter 6 we follow this up with a stochastic version of DisPeL to explore how randomisation in DisPeL itself can be used to enhance its performance.

Chapter 6

Exploiting Randomisation in DisPeL

6.1 Introduction

DisPeL is a deterministic algorithm: there are no inbuilt random decisions and it is guided by a set of fixed rules implemented in a fixed order. These affect the algorithm in a way that makes it vulnerable to the effects of "bad" random initialisations; where determinism can lock the algorithm onto paths that keep the search wandering about, for long periods, in unprofitable regions of the search space. This behaviour is evident from the RLD plots in Chapter 5 which showed that on the same problem, DisPeL could find a solution with less than 100 iterations on some runs and required about 10,000 iterations on others. In the worst case, it may not find a solution given its incompleteness.

In this chapter, we consider a modification to DisPeL that introduces a stochastic element into a critical part of its deadlock resolution strategy - giving it opportunities to alter the search trajectory in a non-deterministic manner as a search progresses. We show that this randomisation can boost performance while reducing its memory requirements, as well as the complexity of the deadlock resolution process.

This chapter is structured as follows. First we briefly review some strategies for exploiting randomisation in combinatorial search in Section 6.2. The new version of DisPeL

is presented in Section 6.3 and in Section 6.4 we discuss how an optimal values for its critical parameter were established. Finally, we present results of empirical experiments where we compare the new algorithm with DisPeL and the Distributed Stochastic Algorithm in Section 6.5.

6.2 Exploiting randomisation in combinatorial search

Hoos and Stutzle introduced Run Length Distribution plots, in [55], to analyse the run time behaviour of Stochastic Local Search (SLS) algorithms; while studying how an algorithm's performance varied with random initialisations and inbuilt random decisions. The distribution of search costs, in their study, showed that certain initialisations led to solutions with short runs irrespective of algorithm parameter settings. They argue that although SLS algorithms are approximately complete as run time approaches infinity, search efficiency actually decreases over time. Therefore, the effects of 'good random initialisations' can be exploited to boost performance of algorithms with periodic restarts from new random points. Their experiments 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 increasing the probability of finding solutions; especially when compared to single runs over longer time spans with the same algorithm.

In related work by Hutter et al [57], on dynamic local search algorithms for solving SAT formulae, randomisation for undoing the effects of cast landscape modifications was investigated. A scheme was introduced where, with a small probability, weights on constraints are smoothened towards the average of all constraint weights. It was demonstrated that randomisation can be used to reduce the complexity of weight update procedures and still allow the underlying weighted hill-climber to outperform the more complicated algorithms - notably the Exponential Sub-Gradient algorithm [100].

A similar study on complete backtracking algorithms by Gomes et al [38] also show benefits of randomisation in search. The study focused on the effects of randomisation in tie-breaking decisions and it was argued that when the heuristic evaluation of which variable to label next was equal for two or more variables, the random selection of one

affects the run time of the algorithm to the extent that run time becomes highly variable and unpredictable. Hence, on some runs the same algorithm may find a solution in seconds and yet on others, it may not find a solution in hours. In their strategy for exploiting randomisation, a new heuristic evaluation function for determining the next variable to label is introduced which increases the number of choices at each branching point and hence the number of random decisions made at different points. In addition, periodic restarts from the root of the search tree were also introduced. The combination of these modifications boosted performance of the underlying algorithms with speed-ups of several orders of magnitude.

A handful of similar randomisation strategies have been explored in the literature of distributed constraint reasoning. In the Distributed Stochastic Algorithms (DSA), presented in [128], all agents are active in parallel and each agent decides randomly whether to improve the solution or to do nothing in each iteration. In addition to reducing the amount of incoherence in their decisions, the random choice to do nothing also helps the algorithm to avoid local optima. Extensions of DSA, in [5], introduced additional randomisation in the form of a random choice to make uphill moves when an agent has no improvements available. Furthermore, the authors also introduced a version of the algorithm where the probabilities for agents to make uphill moves decay over time fashioned after the centralised simulated annealing algorithm [60].

Similarly, non-deterministic tie-breaking schemes were proposed for DBA in [119], which allowed agents to occasionally override their coordination heuristic permitting some connected agents to change values simultaneously and, at the same time, make tie-breaking non-deterministic.

6.3 Stochastic DisPeL

As the RLDs in Chapter 5 show, DisPeL can undoubtedly benefit from a periodic restart strategy with new random instantiations. But the main drawback with periodic restarts is that it is difficult to automatically determine appropriate cut-offs a priori. And since the performance of a restart strategy is particularly sensitive to the cut-off, one may need

to carry out a lot of experimentation before hand to find optimal cut-offs for different problem types and sizes.

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. Therefore, we change the agents' behaviour so that whenever an agent is at a quasi-local-minimum, it decides randomly either to perturb the solution (with probability p) or to increase incremental penalties (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 Stochastic Distributed Penalty Driven Search (Stoch-DisPeL), and implement its new stochastic behaviour by replacing the **check_for_deadlock()** procedure listed in Algorithm 5.2 with the one outlined in Algorithm 6.1. 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. Agents at the receiving end will still act in the same deterministic manner of prioritising the incremental penalty requests over temporary penalty requests.

Algorithm 6.1 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
```

6.4 Determining an optimal p value

The new parameter (p) has a significant impact on Stoch-DisPeL's performance; the simple act of randomly determining which type of penalty is used or how often the temporary penalty is used influences the speed of deadlock resolution and determines the trajectories the search follows as it progresses. In Section 5.4.3 it was shown that DisPeL relied on the temporary penalty to resolve deadlocks i.e. it was used in at least 60% of the time. This appears to work well on the different classes of problems DisPeL was tested on. However, for Stoch-DisPeL, we found that the impact of p in Stoch-DisPeL varies from one problem class to another - on some problem classes it is beneficial to use the temporary penalty more often (i.e. a large p) and on others the reverse is the case. In the following, we present empirical studies of how the value for the parameter affects the algorithm's performance on classes of unstructured problems (Section 6.4.1) and structured problems (Section 6.4.2).

6.4.1 Impact of p on unstructured problems

For unstructured problems, we used random DisCSPs and distributed graph colouring instances for the investigation. Two sets of experiments were run on each problem class. First, multiple runs on single instances were used for RLD analysis and, secondly, promising values of p from the first experiments were used for other experiments with a larger problem set.

In the RLD plots in Figures 6.1 and 6.2, we ran Stoch-DisPeL on single problem instances to compare the effect of the different values of p from 0.1 to 0.9 (in increments of 0.1). In all cases we started the runs from the same random initialisation, so that the only influence on performance was the random choice made when deadlocks are encountered (i.e. p). In Figure 6.1, the plots[1] 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 6.1.

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

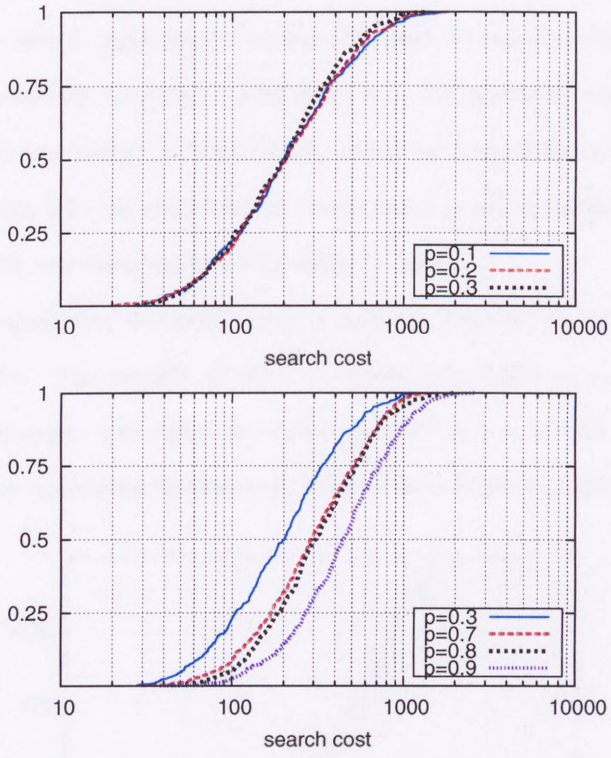

Figure 6.1: Run Length Distribution of Stoch-DisPeL on a distributed graph colouring instance with different values for p.

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 6.1: Average and median search costs in Stoch-DisPeL from RLD analysis in Figure 6.1, for different values of p.

The plots in Figure 6.1 show that while the average cost in Table 6.1 varies, the performance of the algorithm is almost identical for values of p from 0.1 to 0.4. and the median costs are also comparable. Pairwise Student t-tests for the values show that the distributions are mostly identical. And from $p = 0.5$ onwards search costs increase steadily. What stands out distinctly from the results is the huge difference between $p = 0.8$ and $p = 0.9$ (see Figure 6.2), the increase in search costs is abrupt and the probability of finding a solution with any time limit is significantly lower.

The same experiment was repeated using a random DisCSP instance ($\langle n = 60, d = 15, p1 = 0.1, p2 = 0.6\rangle$). The results, plotted in Figure 6.2, follow a similar pattern with the earlier ones. Although, this time the differences from $p = 0.7$ to $p = 0.9$ are more distinguishable. There are sharp increases in search costs from one value to the next.

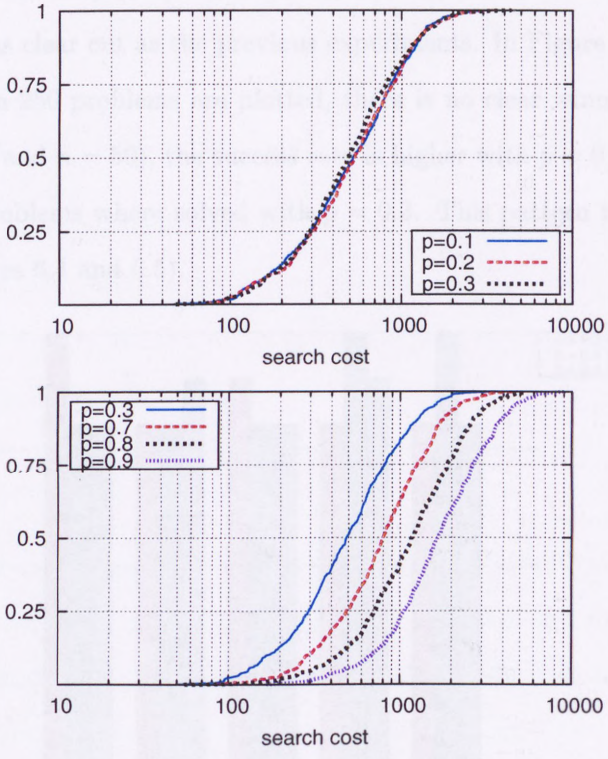

Figure 6.2: Run Length Distribution of Stoch-DisPeL on a random DisCSP instance with different values for p.

Based on the results presented here, it was hard to put a finger down on a best value for p. It is clear, however, that performance is less than optimal for large values of p; and

p	average cost	median cost
0.1	648.8	522.5
0.2	657.0	535.0
0.3	625.4	488.0
0.4	693.4	581.0
0.5	724.3	599.0
0.6	816.4	687.5
0.7	967.0	797.0
0.8	1413.9	1174.0
0.9	2052.6	1675.0

Table 6.2: Average and median search costs in Stoch-DisPeL from RLD analysis in Figure 6.2, for different values of p.

any value between 0.1 to 0.4 was equally good. To confirm this, a second experiment was carried out where we tested the algorithm on a larger dataset with problems of different sizes, using three values for p (0.2, 0.3, 0.4). The results, which are shown in Figures 6.3 to 6.5, are also not as clear cut as the previous experiments. In Figure 6.3, where success rates on attempts on 250 problems are plotted, there is no clear winner - in two groups of problems ($n = 30$ and $n = 50$), the success rate is higher with $p = 0.2$. And in another two groups, more problems where solved with $p = 0.3$. This pattern is also exhibited in the cost plots (Figures 6.4 and 6.5).

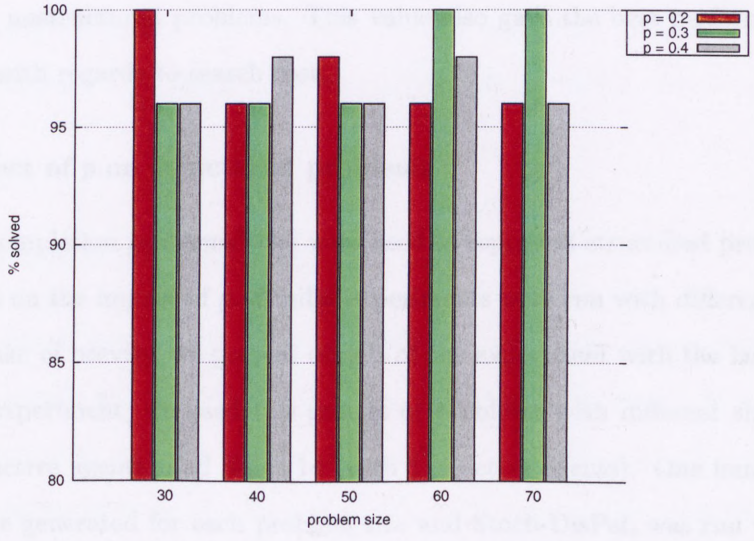

Figure 6.3: Percentage of random DisCSPs solved by Stoch-DisPeL with different values for p.

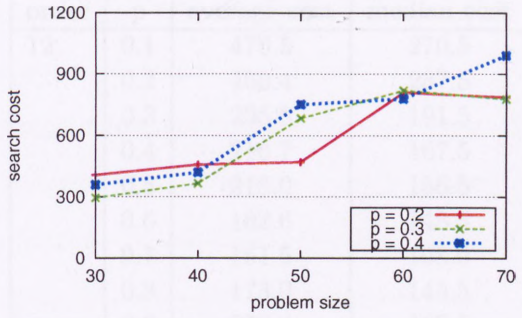

Figure 6.4: Average search costs from runs in Figure 6.3.

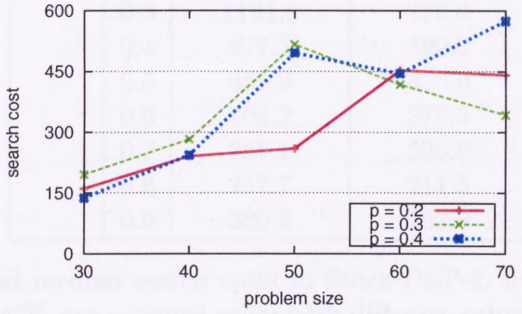

Figure 6.5: Median search costs from runs in Figure 6.3.

The results from all experiments carried were not clear cut, therefore we chose arbitrarily to run Stoch-DisPeL with a probability of 0.3 in the rest of our evaluations of the algorithm on unstructured problems. This value also gave the best performance in both experiments with regards to search cost.

6.4.2 Impact of p on structured problems

Quasigroup completion problems[2] [36] were used to represent structured problems for our investigations on the impact of p. Similar experiments were run with different values of p, but for the sake of brevity, we present details of the experiment with the larger dataset.

For this experiment, we used two groups of problems with different sizes i.e. order 12 (with 84 active agents) and order 14 (with 114 active agents). One hundred solvable instances were generated for each problem size and Stoch-DisPeL was run with different values for p on each set for a maximum of 14,400 and 19,600 iterations respectively.

[2]These are described formally in Section 6.5.3.

order	p	average cost	median cost
12	0.1	476.5	270.5
	0.2	460.4	287.5
	0.3	295.8	191.5
	0.4	216.7	167.5
	0.5	216.0	156.5
	0.6	182.6	148.5
	0.7	161.5	108.0
	0.8	173.0	145.5
	0.9	259.4	162.5
14	0.1	3507.1	2466.5
	0.2	2018.7	1202.5
	0.3	1121.8	776.0
	0.4	977.2	590.0
	0.5	455.9	313.0
	0.6	404.2	278.5
	0.7	267.1	200.0
	0.8	257.7	211.5
	0.9	380.3	263.0

Table 6.3: Average and median search costs in Stoch-DisPeL from runs on quasigroup completion problems (42% pre-assigned cells) with different values of p.

The results summarised in Table 6.3 are quite conclusive, they clearly show that on both problem sizes the minimum median costs were achieved with a value $p = 0.7$. On the smaller problems, the minimum average was found with the same value. While on the larger instances, a slightly better minimum average was found at $p = 0.8$. Nevertheless, for later experiments on structured problems, we used 0.7 for all runs irrespective of the problem size.

6.5 Empirical Evaluation

In further empirical evaluations of Stoch-DisPeL, we compared its performance with DisPeL as well as with a modified version of the Distributed Stochastic Algorithm (DSA). DSA is a distributed iterative improvement search algorithm that relies completely on stochastic decisions to avoid deadlocks. In DSA, all agents act concurrently selecting and exchanging new assignments. 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 retain its current variable assignment (with probability $1 - \alpha$) - where α is

the probability of parallelism. In addition, deadlocked agents are also permitted to make sideways moves that do not worsen their evaluations.

DSA was initially designed to solve optimisation problems - distributed scan scheduling problems [27] in particular, where the agents are expected to reside on simple interconnected devices with limited computational resources. Given the nature of the problem, the algorithm was designed to allow agents to satisfy as many constraints as quickly as possible, rather than to find zero cost solutions. And, it has been shown to converge quicker to locally optimal solutions than DBA[127]. However, Hirayama and Yokoo, in [49], point out that because DSA has no explicit mechanism for escaping from local optima it has low success rates in decision problems where the goal is to satisfy all constraints. Once stuck, the sideways moves are usually not enough to push a search out of locally optimal regions and hence, the algorithm will have difficulty in solving many problems.

Later work by Arshad and Silaghi [5] introduced additional randomisation allowing agents to make uphill moves with a probability ($p2$). Their modified version, called DSA-B1, enabled DSA to find paths out of plateaus in the cost landscape. The authors extended this further in the Distributed Simulated Annealing (DSAN) algorithm where $p2$ decays overtime. We modified DSA-B1 (to make DSA-B1N) to prevent agents from making uphill moves whenever they had consistent assignments, giving agents the opportunity to make "informed" random decisions. We found that this improved the performance of the algorithm significantly because by stabilising things it allowed for increased search intensification activity.

We also found that DSA-B1N was stronger than DSAN. The fixed value of $p2$ meant that the algorithm still had opportunities to find solutions as time went on; whereas, with DSAN we observed that as $p2$ gets smaller the probability of the algorithm finding a solution decreases as well because there are fewer opportunities to make the moves needed to escape from plateaus. Therefore, in the experiments reported in this chapter 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[3] and with structured problems). We use these settings for experiments with DSA-B1N reported here.

We compared the algorithms (Stoch-DisPeL, DisPeL, and DSA-B1N) on random DisCSPs, SAT formulae from the SATLib dataset [56][4], and structured DisCSPs. 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.

6.5.1 Performance on Random DisCSPs

Random binary DisCSPs of different sizes ($30 \leq n \leq 100$) where 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 6.6, 6.7, and 6.8, showing the success rates, median and average search costs respectively.

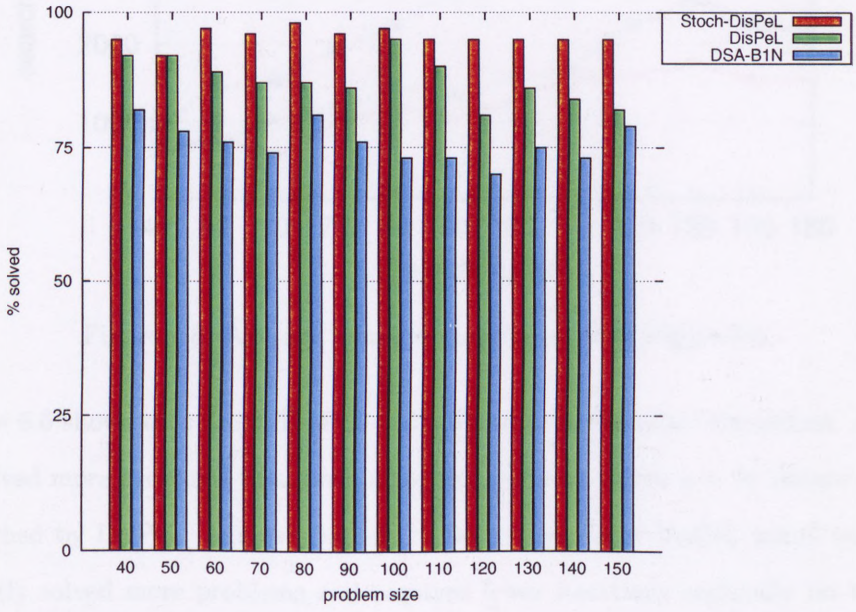

Figure 6.6: Percentage of random DisCSPs solved by Stoch-DisPeL, DisPeL, and DSA-B1N.

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

[4]The instances are available online at http://www.satlib.org.

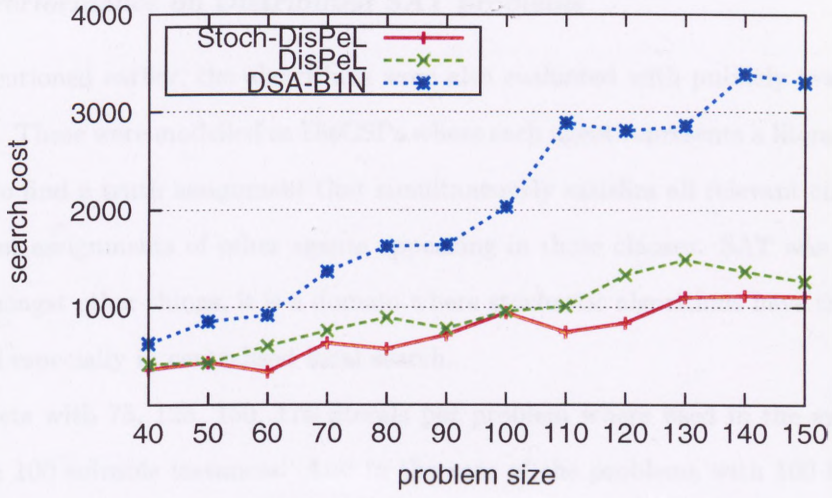

Figure 6.7: Median search costs from runs in Figure 6.6.

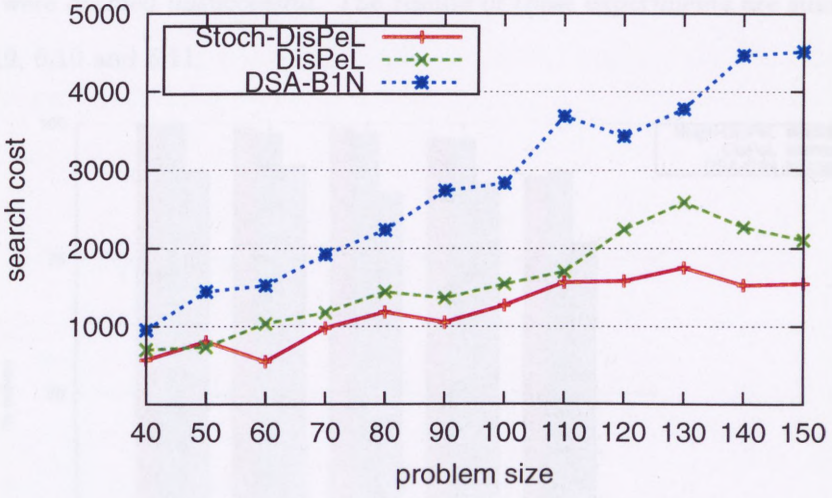

Figure 6.8: Average search costs from runs in Figure 6.6.

Figure 6.6 shows that Stoch-DisPeL is dominant over the other algorithms. It consistently solved more problems than both algorithms, except in the $n = 50$ dataset where it was matched by DisPeL. Against DSA-B1N, both versions of DisPeL faired well. They consistently solved more problems and required fewer iterations especially on the larger problems (Figures 6.7 and 6.8).

6.5.2 Performance on 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 satisfies 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 fared 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 in the case of the problems with 100 literals, the first 500 instances from the dataset were used. 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 6.9, 6.10 and 6.11.

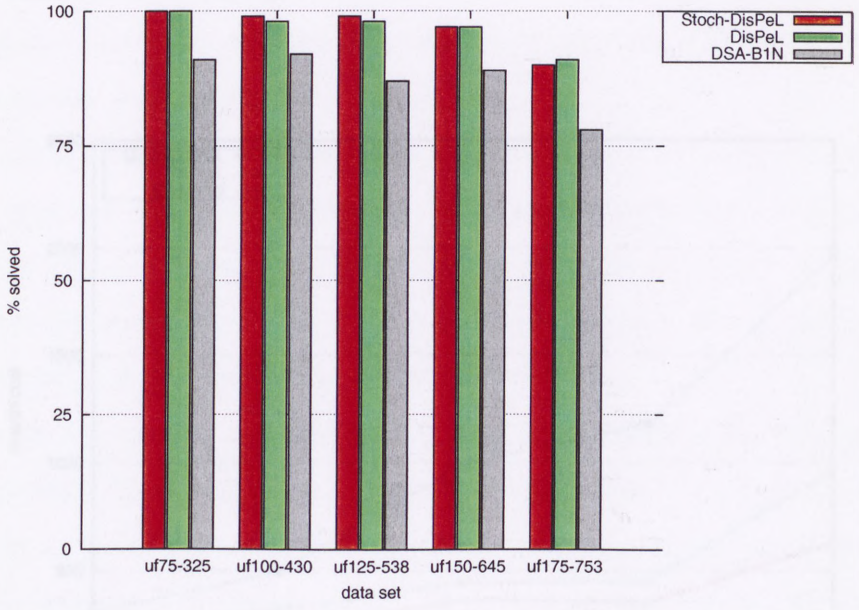

Figure 6.9: Percentage of problems solved by Stoch-DisPeL, DisPeL, and DSA-B1N from attempts on benchmark SAT instances.

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

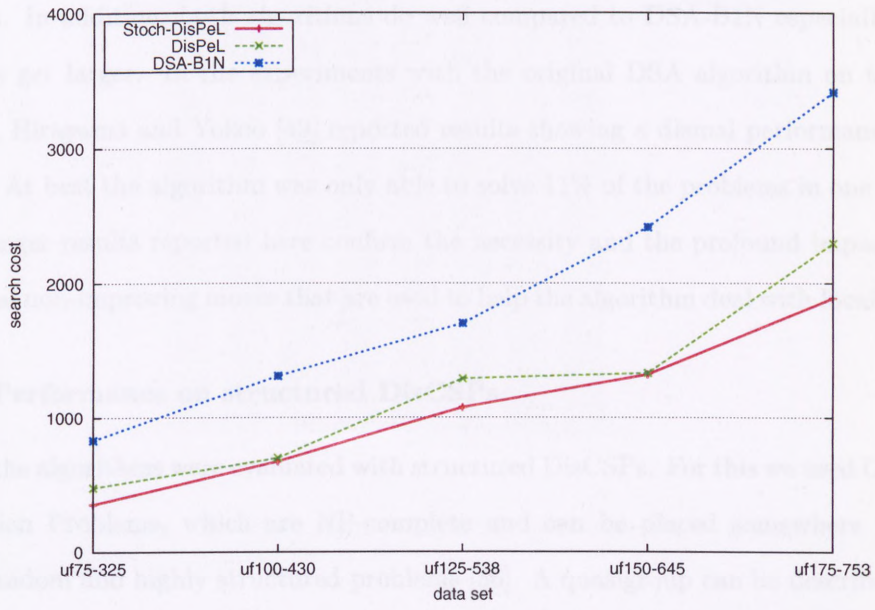

Figure 6.10: Average costs (iterations) of Stoch-DisPeL, DisPeL, and DSA-B1N used-up to solve the problems in Figure 6.9.

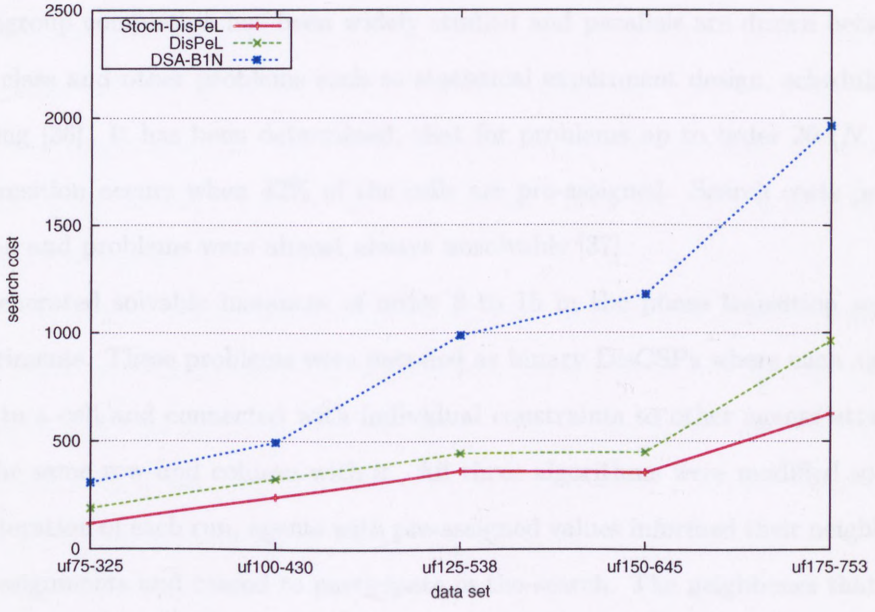

Figure 6.11: Median costs (iterations) of Stoch-DisPeL, DisPeL, and DSA-B1N used-up to solve the problems in Figure 6.9.

this criterion, the results are fairly consistent with those from the experiments on random DisCSPs. In addition, both algorithms 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 [49] 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.

6.5.3 Performance on structured DisCSPs

Finally, the algorithms were evaluated with structured DisCSPs. For this we used Quasiqroup Completion Problems, which are NP-complete and can be placed somewhere between purely random and highly structured problems [36]. A quasigroup can be described as an N by N matrix in which N elements are placed in its cells, such that an element occurs exactly once each row and exactly once in each column. In quasigroup completion, some cells have pre-assigned elements and the problem is to determine if the empty cells can be filled to complete a quasigroup[5] [36].

Quasigroup completion has been widely studied and parallels are drawn between the problem class and other problems such as statistical experiment design, scheduling, and timetabling [36]. It has been determined, that for problems up to order 20 ($N \leq 20$) a phase transition occurs when 42% of the cells are pre-assigned. Search costs peaked at this region and problems were almost always unsolvable [37].

We generated solvable instances of order 8 to 15 in the phase transition region for our experiments. These problems were encoded as binary DisCSPs where each agent was assigned to a cell and connected with individual constraints to other agents attached to cells in the same row and column with it. All three algorithms were modified so that in the first iteration of each run, agents with pre-assigned values informed their neighbours of the pre-assignments and ceased to participate in the search. The neighbours that receive such messages place huge fixed penalties[6] on the values received from the pre-assigned

[5] You may think of it as a CSP formalisation of Sudoku.

[6] These penalties were used in DSA-B1N as well, and they should not be confused with the incremental

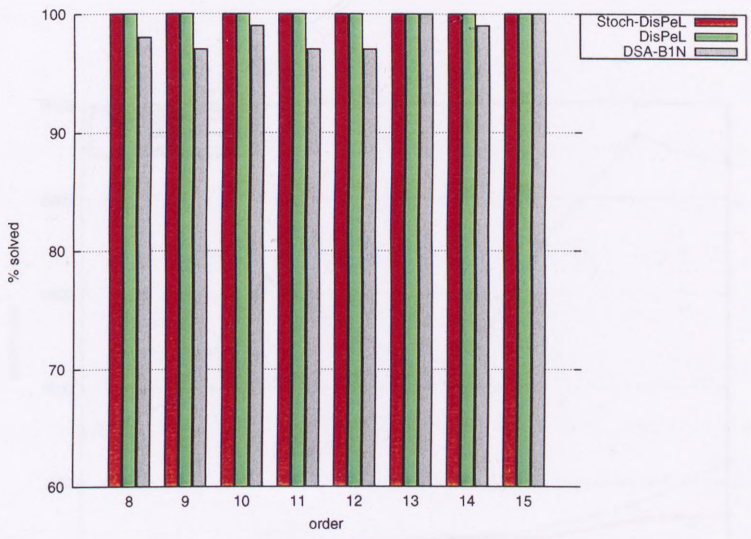

Figure 6.12: Percentage of quasigroup completion problems ($N \times N$ agents) solved by Stoch-DisPeL, DisPeL, and DSA-B1N .

agents, to have the same effect as domain reduction in complete search. Furthermore, after "pruning" the affected values, agents also removed the pre-assigned neighbours from their AgentViews. For this problem class, we found that DSA-B1N's performance was optimal with the parameters $p = 0.5$ and $p2 = 0.02$.

100 instances were generated for each problem size. And like all other experiments, the algorithms were limited to a maximum of $100n$ iterations; where n is the number of agents (cells) in each problem. Results of these experiments are presented in Figures 6.12, 6.13, and 6.14 which show the percentage of problems solved, the average, and the median search costs respectively.

On the structured problems, DSA-B1N was unable to solve a handful of instances and its search costs were considerably higher than both Stoch-DisPeL and DisPeL. Cost-wise, Stoch-DisPeL and DisPeL do equally well on the smaller instances but DisPeL's search costs increase at a much faster pace from order 13 onwards. Overall, the results are consistent with those reported in Sections 6.5.1 and 6.5.2.

penalties in DisPeL and Stoch-DisPeL.

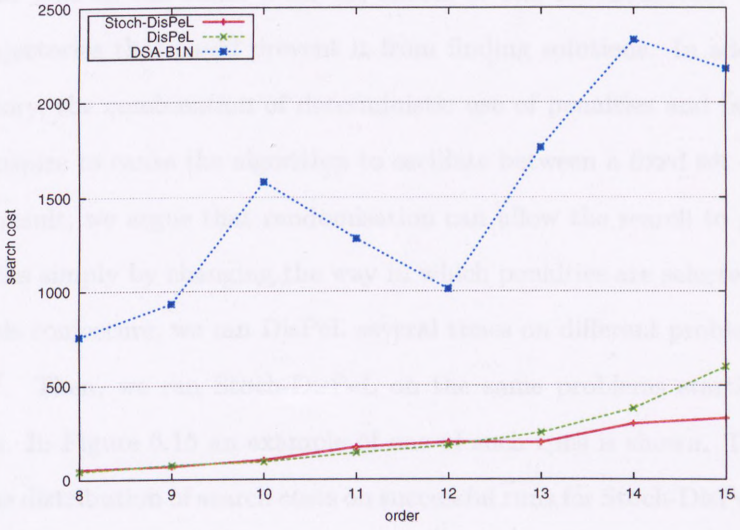

Figure 6.13: Average search costs from successful runs on the quasigroup completion problems.

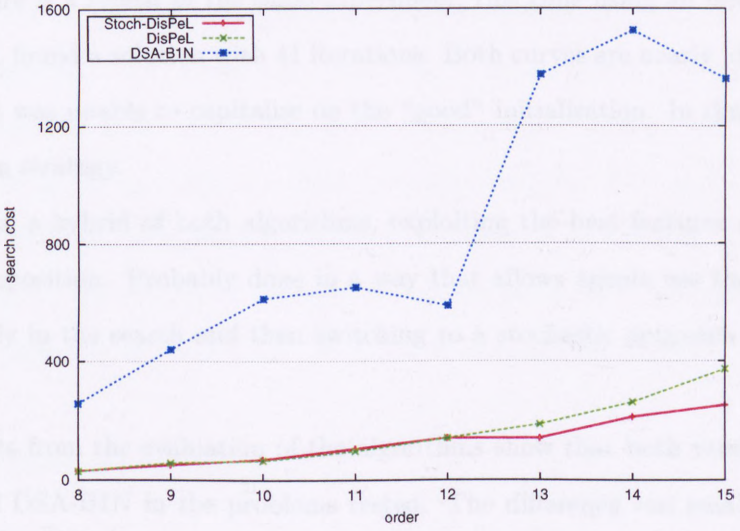

Figure 6.14: Median search costs from successful runs on the quasigroup completion problems.

6.5.4 Discussion

We argue that DisPeL can suffer from the effects of bad initialisations, where it is put on search trajectories that could prevent it from finding solutions. In addition, once on a bad trajectory, the combination of deterministic use of penalties and frequent penalty resets can conspire to cause the algorithm to oscillate between a fixed set of non-solution states. As a result, we argue that randomisation can allow the search to get out of such bad trajectories simply by changing the way in which penalties are selected.

To test this conjecture, we ran DisPeL several times on different problems to find bad initialisation[7]. Then, we ran Stoch-DisPeL on the same problems starting it off these instantiations. In Figure 6.15 an example of one of such runs is shown. The "bad start" curve plots the distribution of search costs on successful runs for Stoch-DisPeL on a random DisCSP ($\langle n = 80, d = 15, p1 = 0.1, p2 = 0.5 \rangle$) 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 it appears not to have suffered from the effects of the bad initialisation. Obviously randomisation is a double edged sword, it can also prevent the algorithm from finding a solution quickly. The "good start" curve in the same figure is a repeat of the same experiment, this time using an initialisation with which DisPeL found a solution with 41 iterations. Both curves are nearly identical, clearly Stoch-DisPeL was unable to capitalise on the "good" initialisation. In that, a risk of the randomisation strategy.

Given this, a hybrid of both algorithms, exploiting the best features of either, is an attractive proposition. Probably done in a way that allows agents use the deterministic approach early in the search and then switching to a stochastic approach as the process draws on.

The results from the evaluation of the algorithms show that both versions of DisPeL outperformed DSA-B1N in the problems tested. The difference was smaller in the Dis-SAT experiments, which as we noted earlier is a domain where stochastic algorithms are expected to do well. DSA, as originally proposed, suffered from an inability to effectively

[7]Where we considered an initialisation as bad when a solution was not found after a maximum of $200n$ iterations.

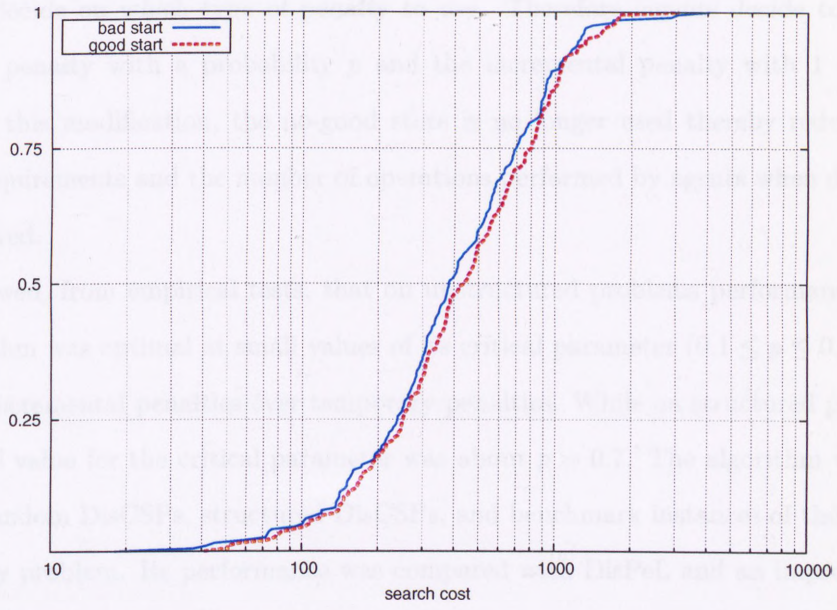

Figure 6.15: Run Length Distribution of Stoch-DisPeL on a problem instance repeatedly starting "good" and "bad" random initialisations.

deal with local optima once stuck; the modifications in DSA-B1N introduced measures to overcome this deficiency. But, the continued reliance on random decisions for these improvements has its drawbacks. For example, the absence of search memory leaves the door open for the algorithm to repeatedly make the same mistakes in attempting to resolve particular deadlocks. But, as we have shown in Table 4.1 albeit in a different context, random unilateral decisions do not quickly resolve as many deadlocks as either the penalty based approach or an approach that uses constraint weights. And, in addition, random moves still caused other constraints, previously satisfied, to be violated. Therefore, deadlocks can linger in the constraint network as a search progresses. Results of DSA-B1N's performance especially in Section 6.5.1 support those earlier assertions.

6.6 Chapter Summary

In this chapter 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 on unstructured problems performance of the new algorithm was optimal at small values of its critical parameter ($0.1 \leq p \leq 0.4$) hence favouring incremental penalties over temporary penalties. While on structured problems, the optimal value for the critical parameter was about $p = 0.7$. The algorithm was evaluated on random DisCSPs, structured DisCSPs, and benchmark instances of the boolean satisfiability problem. Its performance was compared with DisPeL and an improved version of the Distributed Stochastic Algorithm on the same problems. The results showed that randomisation improves performance of the penalty driven strategy; Stoch-DisPeL consistently solved more problems than DisPeL and DSA-B1N, and it typically required fewer iterations in the process.

Chapter 7

Solving coarse-grained DisCSPs

7.1 Introduction

In previous chapters we discussed versions of DisPeL for solving problems where each agent has just one variable, a case for which there are limited realistic scenarios. In this chapter we consider the more realistic scenario where DisCSPs are coarse-grained with agents holding multiple local variables. In such cases, like timetabling or meeting scheduling, besides the constraints between variables held by different agents there are also local constraints between variables within an agent. These kinds of problems can prove to be a real test for collaborative problem solving where agents have to find a balance between the emphasis they place on resolving either the internal or the external constraints. Placing slightly more emphasis on one group of constraints can compromise the collective ability of agents to reach agreement and solve problems.

In this chapter we present two distributed iterative improvement algorithms for solving coarse grained DisCSPs. First, we introduce an extension of Stoch-DisPeL, Multi-DisPeL, for agents with multiple local variables. We also discuss a modification of Eisenberg's Distributed Breakout for coarse-grained DisCSPs where we introduced new heuristics for controlling the growth of constraint weights and some randomisation. We show that these modifications improve the performance of the algorithm considerably.

The chapter is structured as follows. First, in Section 7.2, we brielfy discuss coarse

grained DisCSPs and briefly review the literature on algorithms for solving them. In Section 7.3, we introduce the extension to Stoch-DisPeL, following that we also introduce the modified breakout algorithm in Section 7.4. Finally, in Section 7.5, we present the results of empirical evaluations of both algorithms along with comparisons with other similar algorithms. Finally, in Section 7.6 we briefly highlight some possible extensions/variations for the new algorithm.

7.2 Background

In the model of DisCSP considered so far, we have always assumed that each agent owns exactly one variable, and that in a real-world setting information about each variable resides on a separate machine. At this lowest level of granularity, the amount of information about a problem available to agents is restricted to knowledge about the constraints they are involved in and value updates received from neighbouring agents during the search. As a result, there is a limited amount of computation that agents can perform locally and all the search effort is focused on the distributed collaborative activity.

There are, however, problems that come naturally in a different form / model from that described above i.e. DisCSPs which are made up of interconnected sub-problems. Each sub-problem, naturally distinct from other parts of the problem, is a CSP on its own comprising a set of variables and constraints between those variables, as well as constraints between some variables in the local CSP and variables in other sub-problems (as illustrated in Figure 7.1). Therefore, rather than represent variables, each agent in the DisCSP represents a sub-problem. Distributed lecture timetabling is an example of such coarse grained DisCSPs, where agents represent lecturers and each sub-problem is the set of courses taught by an individual. The constraints in the CSPs include those stating that an individual can not teach two different courses at the same time (intra-agent constraints), as well as constraints to prevent some clashes with courses taught by other lecturers (inter-agent constraints) either because of student course registrations or resource availability.

Agents in these DisCSPs inevitably are more complex, compared to the agents used so

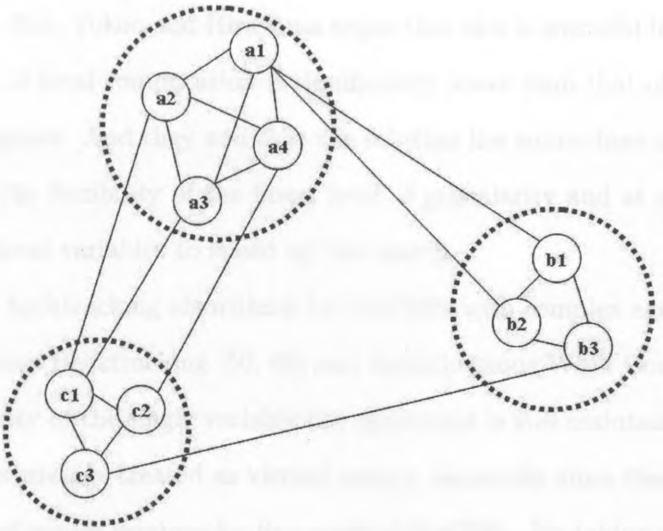

Figure 7.1: An illustrative example of a coarse grained DisCSP, with 3 inter-connected sub-problems / agents.

far, since they each control more than one variable and as such have much more problem information available. Consequently, agents have the opportunity (or are required) to do more local computation in the search for a solution.

According to Yokoo and Hirayama [124], the amount of local computation (and the level of granularity) required by complex agents can vary along two extremes. At one extreme, a problem can be formalised as a fine grained DisCSP where each sub-problem becomes a variable. Therefore, each agent does all its local computation before hand by finding all possible solutions for its sub-problem which are taken as the "domain values" of the new variables. These local solutions are exchanged with other agents during the collaborative search. This approach was adopted in [120] on their work on distributed SAT solving. But, Yokoo and Hirayama argue that the obvious limitation of this approach is that when local sub-problems get large and complex, it may be impossible to find all local solutions initially.

At the other extreme, there is the option of treating each variable in a sub-problem as a virtual agent - in short, running a single variable per agent algorithm, where local computation is minimal and all effort is expended on the distributed search. Therefore, agents simulate all activities of these virtual agents including communications between them and

other real agents. But, Yokoo and Hirayama argue that this is wasteful because, according to them, the cost of local computation is significantly lower than that of communications between virtual agents. And they add that the solution lies somewhere in between, where agents can enjoy the flexibility of the finest level of granularity and at same time exploit the clustering of local variables to speed up the search.

In distributed backtracking algorithms for DisCSPs with complex agents, most prominently Asynchronous Backtracking [50, 69] and Asynchronous Weak Commitment Search [124], the granularity of the single variable per agent case is still maintained and therefore variables are inadvertently treated as virtual agents; especially since these algorithms are direct extensions of earlier versions for fine grained DisCSPs. By taking each variable as a virtual agent, agents in those algorithms use a single strategy to deal with both inter-agent and intra-agent constraints as there is typically no distinction between the constraints. As such, deadlocks are still detected (and no-goods generated) from each variable rather than from entire sub-problems. At the same time, the amount of computation done locally within agents is still significant. Each agent will typically try, exhaustively in the worst case, to find a local solution that is consistent with higher priority external variables before either extending the partial solution or requesting revision of earlier choices by other agents.

7.3 Multi-DisPeL: DisPeL for agents with multiple local variables

7.3.1 Algorithm overview

The Distributed Penalty Driven Search for agents with multiple local variables (Multi-DisPeL) is based on Stoch-DisPeL. It is a distributed iterative improvement algorithm which relies on the penalty driven strategies introduced in Chapter 4 to resolve deadlocks. The key features of the algorithm are:

- As in Stoch-DisPeL, agents take turns to improve an initial random solution over successive iterations and communicate new assignments to their neighbours.

- In each iteration, each agent uses a steepest descent local search to improve the evaluations of its variables.
- When the steepest descent search is stuck, penalties attached to the current assignments of inconsistent variables are used to modify cost landscapes.
- Variables are treated as virtual agents whenever penalties are used where penalties are "sent" from individual variables involved in deadlocks to variables sharing constraints with them.
- In the same vein as above, penalties are imposed by agents on local variables and requests for their implementation are also sent to the owners of external variables connected to the originating variables.
- Like Stoch-DisPeL, agents choose randomly between using a temporary penalty and increasing incremental penalties whenever they decide to implement penalties.

Penalties are used aggressively in Multi-DisPeL, compared to DisPeL and Stoch-DisPeL. We no longer require that agents have to detect that states are unchanged in two successive iterations. As that will slow down, or prevent, resolution of local deadlocks within an agent. Rather, penalties are used as soon as the steepest descent search, which agents use locally to improve the solution, is stuck. And, as a result of this, we ensure that in each iteration agents also have more opportunities to discard incremental penalties.

In each iteration, each agent uses a typical steepest descent local search to minimise the total number of constraints violated by its local variables. We modified the standard steepest descent search for agents in Multi-DisPeL to speed up the algorithm. Rather than computing and implementing the best improvements one after the other, agents make improvements as follows:

1. Find all possible improvements to the solution i.e. all assignments v_i that have the least cost in the domain d_{x_i} of each variable x_i.
2. The best improvements which do not conflict i.e involving unconnected variables, are selected and implemented simultaneously.

3. Ties are broken in favour of variables with the largest number of constraints or the highest lexicographic IDs.

These modifications allow us to speed up the search by reducing the computational costs of the steepest descent search. Rather than throw away the computations carried out, agents can use the opportunity to implement all improvements that can be made in parallel - which, using the standard descent approach, would probably be implemented at a much higher cost since the improvements would have to be recomputed over and over again.

Although we use the steepest descent search locally, for all the experiments reported in this chapter, Multi-DisPeL's structure does not necessarily restrict agents to using this strategy internally. We believe that other heuristics or local search modifications specific to particular problems may be used as alternatives, as long as these algorithms can take into account the information contained in penalties. For example, one may replace the steepest descent search with WalkSAT [102] when solving boolean satisfiability formulae; the penalties on truth values can still be used to drive the variable selection heuristics.

Another implication of the new framework, is that agents need not necessarily be homogeneous. Each agent has the opportunity to use a heuristic of its preference for its steepest descent search, or one best suited to its particular sub-problem as long as penalties form part of the evaluation function and contribute to driving the selected heuristic. We discuss this and other possibilities further in Section 7.6.2.

7.3.2 Agent Behaviour

Much of the activity in Multi-DisPeL takes place within agents although they exchange values and penalty requests amongst themselves. As such, in describing the new algorithm, we focus our attention on describing the actions (outlined in Algorithms 7.1 to 7.7) that take place locally within agents when they are active.

We assume that agents know the owners of variables constrained with their local variables; as such we define an agent's neighbours are those agents whose variables share at least one constraint with the variables belonging to the agent. We also assume that each

agent, as well as each variable, has a unique ID.

At initialisation, agents create an ordering using the Distributed Agent Ordering heuristic with their IDs, as done in DisPeL, whereby they partition their neighbouring agents into higher and lower priority agents. Based on this ordering, agents will therefore treat all variables belonging to higher priority agents as higher priority variables as well as taking variables belonging to lower priority agents as lower priority variables. And in the same fashion as DisPeL, each agent communicates with both sets of neighbours and takes its turn to improve the solution after receiving updates from all higher priority neighbours. During the initialisation process, agents also initialise their local variables with random instantiations and exchange these assignments with their neighbours.

During the search, agents take turns to improve the solution - each of them tries to reduce the number of constraints violated by the local variables they control. When it is an agent's turn to act, it runs a steepest descent local search algorithm (Algorithm 7.2); where it repeatedly seeks out assignments for its local variables that improve the solution until it can no longer find any improvements and there are no opportunities to use penalties. The steepest descent search allows agents to rapidly improve their local solutions and when this search gets stuck (or is successful) agents discard penalties on consistent variables as well as those penalties distorting the cost landscapes of inconsistent variables (Algorithm 7.4, line 3). If the local solution is not consistent, penalties are then used by the agents to resolve any outstanding deadlocks and to allow the steepest descent search to resume.

As mentioned earlier, penalties are used more aggressively in this algorithm. Agents are no longer required to check if variables are at quasi-local-minima (i.e. detect that neighbours' values are unchanged in two successive iterations), as with the local steepest descent search a variable's value may be changed more than once in a single iteration. Therefore, penalties are imposed on local variables as soon as an agent's steepest descent search is stuck. The penalties are still used in the same way as in DisPeL and Stoch-DisPeL, so whichever penalty is selected is imposed on the first inconsistent variable and then on all its internal neighbours. Agents will inform lower priority agents that control other variables connected to the inconsistent variable to impose the same penalties on

those variables when the lower priority agents become active. The choice of what penalty to use i.e. temporary or incremental penalty, is a random one. When an agent's steepest descent search is stuck, it scrolls through each of its inconsistent variables and chooses penalties to impose on those variables' current assignments (Algorithm 7.4, lines 7-16).

The fact that agents do not distinguish between internal and external constraints can cause the steepest descent search to go on indefinitely i.e. an agent keeps trying to find a consistent local solution even though values of external variables prevent it from doing so. To prevent this, when the steepest descent is stuck agents do not impose any "new" penalties on variables whose assignments have changed in the current iteration or have been penalised in the current iteration (Algorithm 7.4, lines 8-10).

The steepest descent search terminates, in each iteration, when the local solution is consistent, no further improvements can be found, or agents can not impose any new penalties on the local variables. When this happens, agents send the new variables' assignments to all affected neighbours, as well as any requests to impose penalties.

7.3.3 Theoretical Properties

Since Multi-DisPeL is based on similar ideas to DisPeL and Stoch-DisPeL, Multi-DisPeL shares some theoretical properties with these algorithms. Therefore, like DisPeL, Multi-DisPeL is sound and it terminates if and only if a solution is found. And in the same vein, there are no completeness guarantees since in Multi-DisPeL the search memory is short-lived and therefore can not be used to permanently rule out previously visited regions. Multi-DisPeL's space requirements are also minimal, any additional information agents hold are related to either their variables' domain values (e.g. the penalty vectors) or to their variables (e.g. tokens to detect that a variable has been penalised in the current iteration). Hence, the space complexity increases only linearly with the problem size.

Algorithm 7.1 Multi-DisPeL: Main loop

```
1: initialise
2: ordering ← empty
3: for i = 0 to OwnVars.size do
4:     ordering ← ordering ∧ var_i
5: end for
6: loop
7:     messages ← accept()
8:     while active do
9:         for i = 0 to num(messages) do
10:            processMessage(message_i)
11:        end for
12:        for i = 0 to Vars.size do
13:            if var_i is consistent then
14:                reset var_i.incrementalPenalties
15:            else if cost function for var_i is distorted then
16:                reset var_i.incrementalPenalties
17:            end if
18:            if var_x.penaltyStatus ≠ null then
19:                implement penalty on var_x
20:            end if
21:        end for
22:        improveSolution()
23:        send variable assignments and penalty requests to all neighbouring agents
24:    end while
25: end loop
```

Algorithm 7.2 procedure improveSolution()

```
1: for i = 0 to Vars.size do
2:     x_i.moved ← FALSE
3: end for
4: while true do
5:     improvements ← getImprovements()
6:     if improvements = ∅ then
7:         penaltyImposed ← imposePenalties()
8:         if ¬penaltiesUsed then
9:             break
10:        end if
11:    else
12:        for i = 0 to improvements.size do
13:            x ← impprovements_i.var
14:            update assignment ⟨x, improvements_i.value⟩
15:            x.moved ← TRUE
16:        end for
17:    end if
18: end while
```

Algorithm 7.3 getImprovements()

```
1: impSet ← ∅
2: for i = 0 to Vars.size do
3:     get v ∈ d_i with the minimum h(x_i)
4:     δ ← h(x_i.currentValue) − h(x_i.v)
5:     if δ > 0 then
6:         impSet ← impSet ∪ improvement(x_i, v, δ)
7:     end if
8: end for
   bestImprovements ← ∅
9: for all (improvement, x_i, v_i, δ_i) ∈ impSet do
10:    remove ← FALSE
11:    for all (improvement, x_j, v_j, δ_j) ∈ impSet do
12:        if ¬isNeighbour(x_i, x_j) then continue
13:        if δ_i < δ_j then remove ← TRUE
14:        if δ_i = δ_j then
15:            if (x_i.numConstraints < x_j.numConstraints) ∨ (x_i.id > x_j.id) then
16:                remove ← TRUE
17:            end if
18:        end if
19:    end for
20:    if ¬remove then
21:        bestImprovements ← bestImprovements ∪ improvement(x_i, v_i, δ_i)
22:    end if
23: end for
24: return bestImprovements
```

Algorithm 7.4 imposePenalties()

```
1: penaltyImposed ← FALSE
2: for i = 0 to Vars.size do
3:     if isConsistent(x_i) ∨ cost function of x_i is distorted then
4:         x_i.resetIncrementalPenalties
5:     end if
6: end for
7: for i = 0 to Vars.size do
8:     if x_i.moved ∨ isConsistent(x_i) ∨ (x_i.penaltyStatus ≠ null) then
9:         continue
10:    end if
11:    r ← random value in [0..1]
12:    if r < p then
13:        x_i.penaltyStatus ← sentAddTempPenalty
14:    else
15:        x_i.penaltyStatus ← sentIncreaseIncPenalty
16:    end if
17:    penaltyImposed ← TRUE
18:    implementLocalPenalties(getPenaltyRecipients(x_i), x_i.penaltyStatus)
19: end for
20: return penaltyImposed
```

Algorithm 7.5 getPenaltyRecipients(var_x)

```
1: recipientList ← empty
2: for i = 0 to var_x.constraints.length do
3:     if var_x.penaltySent = increaseIncPenalty then
4:         recipientList ← recipientList ∧ var_x.constraints_i.neighbours
5:     else if constraintViolated(var_x.constraints_i) then
6:         recipientList ← recipientList ∧ var_x.constraints_i.neighbours
7:     end if
8: end for
9: return recipientList
```

Algorithm 7.6 procedure implementLocalPenalties(*recipientList*, *penaltySent*)

```
1: for each x_i ∈ recipientList.size ∩ Vars do
2:     if penaltySent = addTempPenalty then
3:         if x_i.penaltyStatus = null then
4:             x_i.penaltyStatus ← imposeTempPenalty
5:             impose temporary penalty on x_i.currentValue
6:         end if
7:     else
8:         if x_i.penaltyStatus = null ∨ x_i.penaltyStatus = imposeTempPenalty
           then
9:             x_i.penaltyStatus ← increaseIncPenalty
10:            increase incremental penalty on x_i.currentValue
11:        end if
12:    end if
13: end for
```

Algorithm 7.7 procedure processMessage(message)

```
1: update AgentView with message.variable, message.value
2: if message.penaltyRequest = null then
3:     return
4: end if
5: for each var_i constrained with message.variable do
6:     if message.penaltyRequest = increaseIncPenalty then
7:         var_i.penaltyStatus ← increaseIncPenalty
8:     else
9:         if v_i.penaltyStatus ≠ increaseIncPenalty then
10:            var_i.penaltyStatus ← imposeTempPenalty
11:        end if
12:    end if
13: end for
```

7.4 Enhancing Distributed Breakout (DisBO) with weight decay and randomisation

The distributed breakout algorithm (DBA) has been extended for coarse-grained DisCSPs by its original authors in the form of Multi-DB [48, 49], which was shown to be particularly effective at solving distributed SAT problems. Eisenberg, in [25], proposed another extension, DisBO, for his work on project scheduling. This version was largely based on DBA's framework but differed in its emphasis on increasing weights only at real local optima. Eisenberg noted DisBO's ability to identify unsolvable problems or difficult parts of a problem and therefore used it in the first phase of a hybrid algorithm where it was combined with distributed backtracking search to solve distributed scheduling problems. We have studied DisBO, and we propose some modifications to it. We show that the modifications improve its overall performance considerably.

DisBO differs from Multi-DB in that it has an additional third cycle for global state detection, since weights are only increased when the search is stuck at real local optima and not at quasi-local-optima. Therefore, in addition to the *improve* and *ok?* cycles, there is a *detect-global-state* cycle, which is used to determine that either a solution has been found, that the maximum number of cycles has been reached, or that the search is stuck at a real local minimum. But, the *detect-global-state* cycle is expensive, in terms of messages sent, because it requires agents to continuously exchange state messages until they have determined that all messages have reached all agents in the network. This was needed to get a snapshot of the state of the entire network without resorting to a global broadcast mechanism where each agent is assumed to know every other agent in the network.

DisBO limits the amount of computation done locally within each agent to allow agents to focus on the collaborative aspect of the problem solving activity. In DisBO, each agent's variables are partitioned into two sets, private and public variables. The private variables are those variables that have no inter-agent constraints attached to them. The bulk of the local computation done by agents in DisBO are with these private variables, where in each improvement phase the agents repeatedly select values for these variables that minimise the weighted constraint violations until no further improvements are possible. The pub-

lic variables, on the other hand, are treated like virtual agents and DBA's coordination heuristic is used to prevent any two public variables (even those within one agent) from changing their values simultaneously; except in the case that the concurrent changes do not cause the constraints between them to be violated.

In our modification to the algorithm (DisBO-wd), the weight update scheme in DisBO was replaced with a modification of the weight decay scheme from Frank's work on SAT solving with local search [30]. This weight decay strategy uses weights much more aggressively than the standard breakout algorithms. Instead of modifying weights only when a search is stuck at local optima, weights on violated constraints are continuously updated after each move. At the same time, weights are also decayed at a fixed rate during the updates to allow the algorithm focus on recent increments. Frank argues that this strategy allows weights to provide immediate feedback to the variable selection heuristic and hence emphasise those variables in unsatisfied clauses. We modified the update rule further, so that weights on satisfied constraints are continuously decayed as well. Therefore, before computing possible improvements in DisBO-wd, agents update their constraint weights as follows:

Weights on violated constraints at time t are computed as $W_{i,t} = (dr * W_{i,t-1}) + lr$

Weights on satisfied constraints at time t are decayed as $W_{i,t} = max((dr * W_{i,t-1}), 1)$

where:

dr is the decay rate ($dr < 1$).

lr is the learning rate ($lr > 0$).

Several values for these parameters were tested in empirical investigations and we found that DisBO-wd's performance was optimal on distributed SAT problems with the parameters set to $dr = 0.99$ and $lr = 1$[1], which were consistent with the findings in [30]. And, DisBO-wd performance was optimal with the parameters $dr = 0.99$ and $lr = 8$ on

[1]Results of this investigation are presented in Appendix A

random DisCSPs.

With the new weight update scheme, we were able to reduce the number of DisBO's cycles from three to two, since it was no longer necessary to determine if the search was stuck at real local optima. And while doing this, we also moved the termination detection mechanism into the *ok?* cycle, thus bringing it in line with the original distributed breakout framework.

We also included some randomisation in the algorithm, where the coordination heuristic was replaced with the random break from Wittenberg's work on randomising DBA [119] which is used for non-deterministic tie-breaking. In DBA when two neighbouring variables have the same improvement, the variable with the smaller lexicographic ID is given priority to make its change. But, with the random break, in each improvement cycle agents select and communicate random tie-breaking numbers for each variable, and when there is a tie the variable with the lower number is given priority. Therefore, tie breaking is not always in one direction [49].

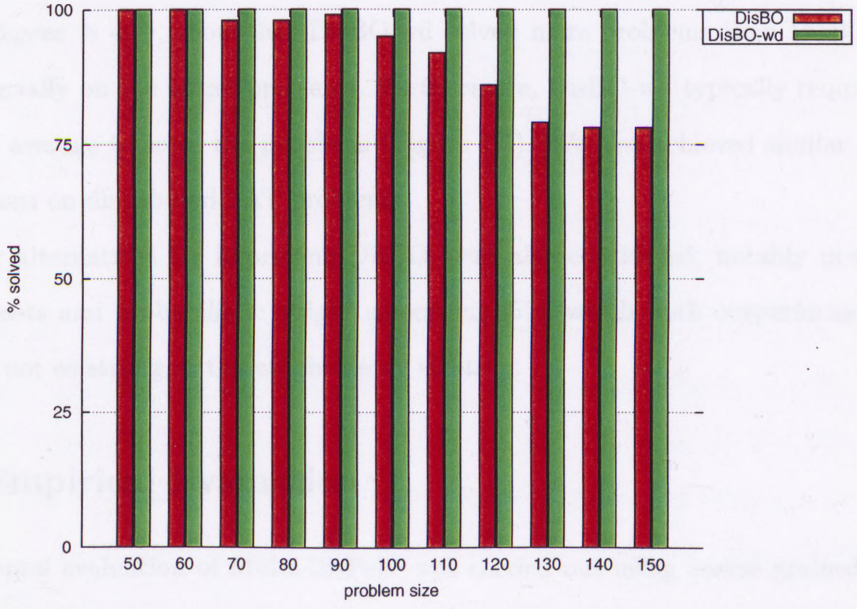

Figure 7.2: Comparison of success rates of DisBO and DisBO-wd on random distributed graph colouring problems of various sizes. Each point represents attempts on 100 problems.

In Figures 7.2 and 7.3, we summarise empirical results from experiments comparing DisBO with our modified version (DisBO-wd). The experiments, which evaluated

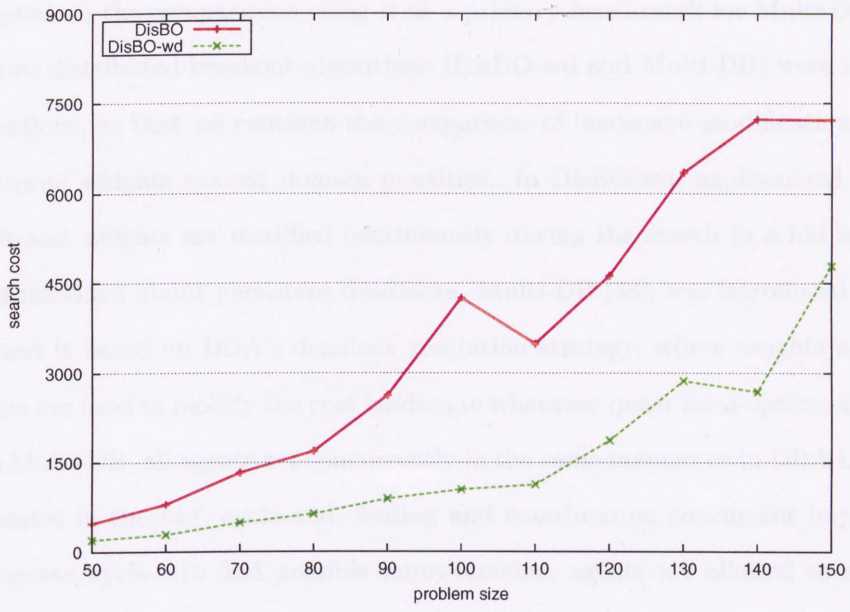

Figure 7.3: Average cycles required by DisBO and DisBO-wd to solve the problems in Figure 7.2.

the algorithms' performance on critically difficult distributed graph colouring problems ($k = 3, degree = 4.7$), show that DisBO-wd solved more problems than DisBO (Figure 7.2), especially on the larger problems. Furthermore, DisBO-wd typically required fewer cycles on average to solve the problems (Figure 7.3). We also achieved similar results in comparisons on distributed SAT problems.

Other alternatives for improving DisBO were also considered; notably probabilistic weight resets and probabilistic weight smoothing [57], which both outperformed DisBO but were not as strong as the weight decay strategy.

7.5 Empirical Evaluation

Experimental evaluation of Multi-DisPeL was carried out using coarse grained versions of several DisCSPs including distributed graph colouring, boolean satisfiability formulae (SAT), and randomly generated DisCSPs. In each case, the algorithm's performance was compared to two versions of Distributed Breakout (Multi-DB and DisBO-wd) and the Asynchronous Weak Commitment Search algorithm (Multi-AWCS)[124]. We also include

Stoch-DisPeL in the comparisons using it as a primary benchmark for Multi-DisPeL.

The two distributed breakout algorithms (DisBO-wd and Multi-DB) were included in the evaluations, so that we continue the comparison of landscape modification strategies i.e. constraint weights against domain penalties. In DisBO-wd, as discussed in Section 7.4, constraint weights are modified continuously during the search in a bid to keep the heuristics informed about persistent deadlocks. Multi-DB [48], was introduced by DBA's authors and is based on DBA's deadlock resolution strategy, where weights attached to constraints are used to modify the cost landscape whenever quasi-local-optima are encountered. In Multi-DB, all agents act concurrently in the same manner as in DBA i.e. sending value updates in the *ok?* cycle and, finding and coordinating concurrent improvements in the *improve* cycle. To find possible improvements, agents are allowed to run a local search algorithm for a fixed number of steps to identify a set of local changes that reduce the cost of the current solution. These changes are exchanged with neighbouring agents, and those changes on variables with the best improvements are accepted; any ties are broken deterministically in favour of the agent with the lowest lexicographic ID. However, simultaneous value changes of two or more co-constrained variables are permitted where such changes do not increase the cost of the solution. Multi-DB works particularly well on distributed SAT problems, and it has been shown to outperform Multi-AWCS in that domain [48, 69]. Multi-DB is used only in the experiments on distributed SAT problems, whereas DisBO-wd is used in all experiments.

Unlike the breakout algorithms and Multi-DisPeL, the Asynchronous Weak Commitment Search is a complete algorithm and is not built around the idea of resolving deadlocks by modifying cost landscapes. Rather, it is an efficient[2] combination of backtracking and iterative improvement search that deals with deadlocks through a combination of variable re-ordering and storage of explicit no-goods, which help the search avoid combinations of values that can not be part of a solution. While the algorithm has been shown to outperform other distributed backtracking algorithms [124], it has the drawback of possibly requiring an exponential amount of memory in the worst case to store no-goods.

[2]With regards to the number of iterations taken on average to solve a problem.

The same metrics, from Chapters 5 and 6, were used to evaluate the algorithms. Algorithms were compared on the percentage of problems solved within a maximum number of iterations (or cycles)[3]. As usual the number of iterations (of cycles) used was taken as the measure of efficiency; in the case of synchronous algorithms we can infer the number of messages exchanged between agents and approximate the number of constraint checks with the metric.

7.5.1 Creating coarse-grained DisCSPs

Publicly available problem instances, as well as randomly generated problems, were used to evaluate the algorithms in this chapter. These problems were partitioned into evenly sized inter-connected sub-problems for each agent using a simple partitioning algorithm which was designed to ensure that there was always a meaningful cluster of variables within each agent i.e. there are constraints between some of the variables belonging to an agent. Sub-problems for each agent (a_i) were created as follows:

1. First, a randomly selected variable (x_i) that is not already allocated to another agent is allocated to a_i.
2. A variable constrained with x_i is randomly selected and allocated to a_i.
3. The process of randomly selecting one of the variables already allocated to a_i and selecting a random neighbour of the variable for allocation to a_i is repeated until the number of required variables for a_i have been found.
4. In addition, with a small probability (p) a randomly selected variable is allocated to a_i, even if it is not connected with any of a_i's existing variables.

7.5.2 Distributed graph colouring

To start off, we evaluated Multi-DisPeL on distributed graph colouring problems to study the relationship between search costs and the problem size, as well as the influence of agent

[3]Given its completeness and unlimited time, Multi-AWCS is guaranteed to solve all problems used since they all have solutions. But, in this case we are interested in its performance in bounded time.

size (i.e. the number of variables each agent controls) on its performance. Results from runs with Stoch-DisPeL, DisBO-wd, and Multi-AWCS on the same problems were used as a benchmark for Multi-DisPeL's performance.

Critically difficult, but solvable, random distributed graph colouring instances ($k = 3, degree = 4.7$) were used for the experiments. We generated problems of different sizes (i.e. with 100, 150, 200, and 250 nodes per instance) and 100 instances for each problem size. These problem instances were partitioned and solved with different numbers of agents (ranging from 2 to 25) so we could directly study the effect of agent size on performance of the coarse grained algorithms. In the experiments, each algorithm was limited to a maximum of $100n$ iterations in each attempt. However, we use a maximum of $200n$ iterations for DisBO-wd, to count its two cycles (i.e. *improve* and *ok?*) separately and as such give it the same number of opportunities to change variable assignments as the other algorithms. We recorded the number of problems solved within the maximum number of iterations and the average and median number of iterations required. The results are presented in two sets, first we show a comparison of Multi-DisPeL and Stoch-DisPeL in Table 7.1. In Tables 7.2 and 7.3, we compare the performance of Multi-DisPeL with Multi-AWCS and DisBO-wd on the same problems looking at performance where variables are distributed evenly and unevenly amongst agents.

The results in Table 7.1 confirm that problem solving is quicker when agents do additional computation when dealing with coarse-grained DisCSPs, as opposed to treating each variable as a virtual agent. Although both Multi-DisPeL and Stoch-DisPeL solve roughly the same number of problems, Multi-DisPeL has lower average search costs. The results also show that performance for Multi-DisPeL improves as agents control more variables, suggesting that agents are able to exploit the opportunity of more problem information to speed up the search.

In Table 7.2, we summarise the results comparing Multi-DisPeL with other coarse-grained algorithms (Multi-AWCS and DisBO) on the same problems from Table 7.1. Both Multi-DisPeL and Multi-AWCS solved about the same number of problems in each set, but performance for DisBO-wd degrades on the larger problems. It solves fewer problems

algorithm	n	agents	% solved	average cost	median cost
Stoch-DisPeL	100	-	100	236.5	111
	150	-	100	686.4	300
	200	-	99	1878.5	890
	250	-	98	2201.2	1277
Multi-DisPeL	100	2	100	84.5	44
		4	100	102.6	43
		5	100	105.3	58
		10	100	112.1	55
	150	3	100	300.7	110
		5	99	271.3	121
		10	100	291.2	148
		15	100	351.1	135
	200	4	100	804.4	329
		5	100	897.3	324
		10	100	1135.4	373
	250	5	99	1242.6	417
		10	100	1660.5	529
		25	97	1785.7	668

Table 7.1: Performance of Multi-DisPeL and Stoch-DisPeL on distributed graph colouring problems.

and its search costs are considerably higher. In Section 5.6 earlier, we already hinted at some of the reasons why the strategy of modifying landscapes with constraint weights and the technique of limiting the concurrent changes can adversely affect the performance of algorithms built around that framework; these effects are evident in the comparison results shown. The results also show that Multi-AWCS generally has lower average search costs than Multi-DisPeL, but Multi-DisPeL has lower median costs in all but one problem set.

We carried out further experiments on distributed graph colouring where we considered the case where all agents do not necessarily have the same number of variables, making the random problems slightly more realistic. So, we modified the partitioning algorithm to distribute an uneven number of variables to a random number of agents. Using critically difficult graph colouring instances (i.e. $k = 3, degree = 4.7$), we evaluated the performance of Multi-DisPeL, Multi-AWCS, and DisBO-wd on the same problems from Table 7.2 in order to study how the algorithms are affected by the new distribution of variables to agents. The results, which are summarised in Table 7.3, follow similar patterns with the previous experiment with Multi-DisPeL having lower median search costs than Multi-

algorithm	n	agents	% solved	average cost	median cost
Multi-DisPeL	100	2	100	84.5	44
		4	100	102.6	43
		5	100	105.3	58
		10	100	112.1	55
	150	3	100	300.7	110
		5	99	271.3	121
		10	100	291.2	148
		15	100	351.1	135
	200	4	100	804.4	329
		5	100	897.3	324
		10	100	1135.4	373
Multi-AWCS	100	2	100	57.0	35
		4	100	114.9	77.5
		5	100	123.24	91.5
		10	100	162.83	129
	150	3	100	222.9	178.5
		5	100	288.3	198
		10	100	341.0	276.5
		15	100	313.1	223
	200	4	100	563.5	422
		5	100	556.0	431
		10	100	704.4	527
DisBO-wd	100	2	100	966.4	725
		4	100	982.6	606
		5	100	1084.3	690.5
		10	100	1019.8	816.5
	150	3	98	4248.3	2436
		5	97	4376.4	2274
		10	98	4977.1	2482
		15	99	3784.1	2176
	200	4	85	10376.2	6526
		5	82	11726.6	6686
		10	83	10262.0	5956

Table 7.2: Performance of Multi-DisPeL, Multi-AWCS, and DisBO-wd on distributed graph colouring problems.

algorithm	n	agents	% solved	average cost	median cost
Multi-DisPeL	100	**r7.2**	100	125.5	61
	150	**r9.6**	99	304.1	129
	200	**r11.9**	100	1056.0	348
Multi-AWCS	100	**r7.2**	100	120.8	98
	150	**r9.6**	100	303.7	262.5
	200	**r11.9**	100	672.0	557.5
DisBO-wd	100	**r7.2**	100	1099.6	568
	150	**r9.6**	100	3872.3	2401
	200	**r11.9**	89	11432.2	7414

Table 7.3: Performance of Multi-DisPeL, Multi-AWCS, and DisBO-wd on distributed graph colouring problems with a random number of agents and an uneven distribution of variables to those agents. **rx.x** is the average number of agents in a problem set.

AWCS. It appears that the uneven distribution of variables increases search costs, for both Multi-DisPeL and Multi-AWCS, and the costs are similar to the cases where agents hold a small number of variables (cf. Table 7.2).

In summary, the experiments show that Multi-DisPeL is clearly competitive on distributed graph colouring compared to its ancestor, Stoch-DisPeL, and well against both Multi-AWCS and DisBO-wd. Compared to Multi-AWCS, Multi-DisPeL's average search costs were slightly higher, but its median search costs are almost always lower than those for Multi-AWCS. In short, Multi-DisPeL is able to achieve similar levels of performance with Multi-AWCS without the additional overhead of creating new constraints (in form of no-goods) and not breaching privacy by connecting variables that were not previously linked in the original specification of the problems being solved.

7.5.3 Distributed SAT problems

In the second set of experiments, we evaluate the performance of Multi-DisPeL and the benchmark algorithms on distributed SAT problems. Satisfiable 3-SAT instances from the SATLib dataset were used for the experiments; made up of formulae with 100, 125, and 150 literals. These were transformed into coarse-grained DisCSPs with the technique specified in Section 7.5.1. We did not run any experiments with Multi-DB and Multi-AWCS, rather we used results on experiments with the same instances from [48], published by

the algorithms' authors, as benchmarks[4]. All of the other algorithms (i.e. Multi-DisPeL, Stoch-DisPeL, and DisBO-wd) were run once on each instance, and were limited to $100n$ iterations (where n is the number of literals in a formulae) before attempts were recorded as unsuccessful [5]. The results in Tables 7.4, 7.5, and 7.6 show the percentage of problems solved, the average search costs, and the median search costs from the runs.

The results for Multi-DB are for a version with periodic random restarts, which Hirayama and Yokoo in [48] found solved more problems than the original version. Similarly, in the same work, the authors used a version of Multi-AWCS without no-good learning to keep their comparisons with Multi-DB fair. For Multi-DisPeL, we found that performance with the fixed parameter values for the temporary penalty (fixed at 3) and the probability of using the temporary penalty (fixed at 0.3) was less than optimal. And, therefore, we adjusted these parameters as follows: (1) the size of temporary penalty was fixed to 2; and (2) the probability of using the temporary penalty was increased to 50%. These parameter tuning made a huge difference to the algorithm's performance in the SAT domain. In the same regard, the parameters for DisBO-wd were set to $lr = 1$ and $dr = 0.99$ for the domain (cf. Section 7.4).

The results of the experiments on distributed SAT problems as plotted in Tables 7.4 to 7.6 show that both versions of the penalty driven search were very strong especially in terms of cost. Both algorithms (Stoch-DisPeL and Multi-DisPeL) do nearly as well as Multi-DB and DisBO-wd in terms of the percentage of problems solved. In fact, Stoch-DisPeL's performance makes a strong case for using simple virtual agents to solve distributed SAT problems given that computational overhead may be lower in this case.

Compared to Multi-DB, DisBO-wd is very competitive especially on the smaller problems. DisBO-wd solves as many problems as Multi-DB and its search costs are much lower on average. DisBO-wd also has a consistency in its search costs for each problem size that Multi-DB does not match. For example, average search costs in the 150 literal problems increase by about 350% as the number agents increase for Multi-DB. While on the same

[4]Variables are randomly distributed amongst agents in [48], so from each agent's perspective the problems may not be exactly the same.

[5]In [48], Multi-DB and Multi-AWCS were limited to a maximum of $250n$ iterations on their runs.

algorithm	agents	% solved	average cost	median cost
Stoch-DisPeL	-	99.1	626	200
Multi-DisPeL	2	99.2	297	93
	4	98.7	455	118
	5	98.1	487	136
	10	98.7	593	154
	20	97.7	576	145
Multi-DB	2	99.9	886	346
	4	100	1390	510
	5	100	1640	570
	10	99.6	3230	1150
	20	99.7	3480	1390
Multi-AWCS	2	99.9	1390	436
	4	98.7	4690	1330
	5	97.6	6100	1730
	10	96.8	7630	2270
	20	95.0	8490	2680
DisBO-wd	2	100	923	515
	4	100	948	495
	5	100	984	490
	10	99.9	1003	516
	20	99.8	993	510

Table 7.4: Performance of Multi-DisPeL and other algorithms on 1000 random distributed SAT problems with 100 literals distributed evenly amongst different numbers of agents.

algorithm	agents	% solved	average cost	median cost
Stoch-DisPeL	-	99	1074	360
Multi-DisPeL	5	95	874	263
	25	96	911	303
Multi-DB	5	100	2540	816
	25	100	6300	2330
Multi-AWCS	5	87	1.92×10^4	9290
	25	80	2.55×10^4	1.58×10^4
DisBO-wd	5	100	1727	725
	25	100	1686	921

Table 7.5: Performance of Multi-DisPeL and other algorithms on 100 random distributed 125 literal SAT problems.

algorithm	agents	% solved	average cost	median cost
Stoch-DisPeL	-	97	1320	353
Multi-DisPeL	3	97	1367	323
	5	90	829	268
	10	93	1214	292
	15	95	1574	368
Multi-DB	3	100	2180	608
	5	100	3230	1200
	10	96	9030	2090
	15	98	9850	3850
Multi-AWCS	3	81	2.43×10^4	1.11×10^4
	5	67	3.71×10^4	2.61×10^4
	10	61	3.94×10^4	3.60×10^4
	15	61	4.23×10^4	4.17×10^4
DisBO-wd	3	99	2078	874
	5	99	2186	910
	10	99	2054	1012
	15	98	1893	898

Table 7.6: Performance of Multi-DisPeL and other algorithms on 100 random distributed 150 literal SAT problems.

problems, DisBO-wd's average search costs remain within a 15% range of the minimum average without a clear degradation in performance as the number of agents increase. Clearly, while both DisBO-wd and Multi-DB, rely on modifying constraint weights to deal with deadlocks, DisBO-wd is less affected by the distribution of variables to agents.

Multi-AWCS is not as strong as the other algorithms in this domain. It solves the least number of problems and it has the highest search costs. As we pointed out, Multi-DisPeL and Stoch-DisPeL do perform quite well in this domain. Both algorithms have lower average and median search costs, even though the percentage of problems solved is marginally lower than Multi-DB and DisBO-wd.

7.5.4 Random distributed constraint satisfaction problems

Finally, we evaluated the algorithms' performance on random DisCSPs. However, in this experiment Multi-AWCS is only used in the runs with the smallest sized problems. It is well documented (for example in [106, 72, 69]) that Multi-AWCS may require an exponential amount of memory to store no-goods during an attempt to solve a problem. The number of no-goods generated may increase exponentially on large problems, and since each no-

good may be evaluated at least once in each iteration, the length of time to complete each iteration increases dramatically as the search progresses. In our experience with the algorithm, we found that it typically ran out of memory on runs with large problems, especially for DisCSPs with 60 variables or more[6], and the algorithm sometimes required considerable amounts of time to solve even a single instance.

The runs with random DisCSPs are similar to the earlier experiments (Section 6.5.1), this time we used three groups of problems with varying sizes and 100 problems in each group. Likewise, we consider the behaviour of the algorithms as the problem size increases as well as the impact of the number of variables each agent owns. The results of these experiments are summarised in Table 7.7 where we show the percentage of problems solved, and the average and the median iterations from successful runs on attempts on 100 instances for each problem size.

The results in Table 7.7 are fairly consistent with the results of experiments on distributed graph colouring. Both Multi-DisPeL and Multi-AWCS have lower search costs than Stoch-DisPeL and DisBO-wd, and DisBO-wd's performance degrades considerably on the largest problems. But, in this case the search costs for Multi-AWCS increase abruptly as the number of agents in the coarse-grained DisCSPs increase. And, apart from the case where the 50 variables are partitioned amongst 2 agents, Multi-AWCS does worse than Multi-DisPeL.

Average and median search costs for Multi-DisPeL show a steady increase, within each problem size, as the number of agents increases. And at the lowest level of granularity, the average search costs are only 13% lower than Stoch-DisPeL on the largest problems; although the median search costs are much lower. This suggests that there is a case to use virtual agents (and an algorithm like Stoch-DisPeL) when there are just a handful of variables per agent and computational resources for each agent are at a premium. Nevertheless, it is clear that in Multi-DisPeL, agents are able to take advantage of the additional problem information from clustering of variables to shorten the time taken to find solutions.

[6]Experiments were run in a Java environment on a 3Ghz Pentium PC with 1GB of RAM.

algorithm	n	agents	% solved	average cost	median cost
Stoch-DisPeL	50	-	99	771	423
	100	-	94	1319	786
	200	-	98	2425	1287
Multi-AWCS	50	2	100	186	90
		5	100	738	288
		10	98	995	527
Multi-DisPeL	50	2	99	250	109
		5	99	307	124
		10	99	309	146
	100	2	94	611	260
		4	99	905	308
		5	97	856	276
		10	94	928	449
	200	4	97	1209	474
		5	95	1382	534
		10	95	2190	727
		20	94	2137	846
DisBO-wd	50	2	98	1770	951
		5	94	1927	1336
		10	99	1855	1104
	100	2	90	5468	3754
		4	79	5251	3922
		5	83	4996	2922
		10	88	4695	3065
	200	4	57	11778	11482
		5	62	13454	8060
		10	65	16832	14432
		20	57	13289	9544

Table 7.7: Performance of Multi-DisPeL and other algorithms on random DisCSPs ($\langle n, d = 10, p1 \approx 0.1, p2 = 0.5 \rangle$).

7.5.5 Summary of results

Results of the experiments in this chapter are consistent with those presented in Section 5.6, when comparing Multi-DisPeL and DisBO-wd. As we had argued earlier, our penalty based strategy is a more effective diversification scheme for dealing with local optima. The results in this chapter give additional support for this conjecture. In all problem classes, Multi-DisPeL solved more problems than DisBO-wd and it required fewer iterations. But, DisBO-wd is quite competitive compared to Multi-DB, which is the other algorithm that relies on constraint weights to deal with local optima; but unlike DisBO-wd, weights in Multi-DB are allowed to grow unbounded. DisBO-wd's search costs were always lower than those for Multi-DB in the SAT problems where each agent has just a few variables although, not doing as much local computation meant that it could not take advantage of the additional information available when the number of variables per agent was large. Nevertheless, the results do support the proposition that retention of constraint weights can have negative effects on the cost landscape, as argued in [81, 111].

Compared to Multi-AWCS, Multi-DisPeL had higher search costs in the experiments with distributed graph colouring problems but lower median costs. Multi-DisPeL was very competitive in the other problem sets. However, Multi-DisPeL had a much lower space complexity than Multi-AWCS; it does not create any new constraints and no new links are created between unconnected agents. Therefore, in Multi-DisPeL the number of messages sent by each agent in each iteration is fixed; whereas as more links are created in Multi-AWCS traffic increases as problem solving progresses, as well as the amount of processing each agent does.

7.6 Algorithm variations

7.6.1 Pre-processing local sub-problems

Search efficiency in Multi-DisPeL can be improved by taking advantage of the information agents have of local sub-problems to pre-process a problem. And such pre-processing can be used to reduce the search space of problem or to determine if a problem is unsolvable

in the first place. For example, if the size of an agent's local problem is not prohibitive, it can perform a complete tree search on it before participating in the collaborative search. And if the tree search indicates that the local sub-problem is unsolvable, the agent can relax some of its local constraints where possible or inform other agents of the futility of embarking on a collaborative search.

Alternatively, agents can also perform arc-consistency on the local sub-problems before the search begins to filter out values that can not be part of a valid solution. And as with a full search, arc-consistency may also indicate that a sub-problem, and by extension the whole problem, is unsolvable. The combination of tree search and consistency maintenance techniques can be used by agents to give stronger guarantees of local consistency.

7.6.2 Heterogeneous agents

As we mentioned in Section 7.3.1, agents are not necessarily restricted to using a steepest descent search heuristic to improve their local sub-problems in Multi-DisPeL. Especially as agents only exchange information about the assignments for their local variables and the requests to implement penalties. Other heuristics can be used locally as long as penalties can be directly incorporated in the cost functions of such methods and agents can easily determine when to implement new penalties. For example, the case of using either WalkSAT or the memory based Novelty [71] as an alternative is straightforward; the modified cost functions can be used to drive the variable selection heuristic and penalties can be imposed when a fixed number of tries have been completed.

Each agent can also choose a heuristic best suited to its individual sub-problem or its processing capacity. And where agents have certain preferences, local heuristics may be geared towards such preferences. Alternatively, in an exaggerated case, agents can even switch between heuristics during a search process. Most likely in hybrid cases where each agent also has a local learning algorithm and therefore, as a search progresses, the best heuristics for their sub-problems are identified and used more often.

7.7 Chapter summary

We presented Multi-DisPeL in this chapter which extends Stoch-DisPeL for coarse-grained DisCSPs. Like Stoch-DisPeL, Multi-DisPeL is a distributed iterative improvement algorithm that utilises penalties on individual domain values to modify cost landscapes and to deal with local optima. In Multi-DisPeL, each agent takes turns to improve the solution where it runs a steepest descent algorithm locally without distinguishing between internal or external constraints. When this local attempt is stuck penalties are imposed on local inconsistent variables as well as on external variables connected to them in the same fashion with Stoch-DisPeL.

In Section 7.4, we introduced DisBO-wd which is a modification of the variant of Distributed Breakout for a agents with multiple local variables presented in [25]. The weight update mechanism in DisBO was modified, taking a leaf from [30], so that weights on constraints also decay as well. All weights are updated in each iteration irrespective of whether agents are at quasi-local-optima or not. We found that, with the changes, DisBO-wd performed better than DisBO. DisBO-wd solved more problems and it required fewer iterations to solve problems. Compared to Multi-DB, DisBO-wd is quite competitive in the SAT domain. DisBO-wd has much lower complexity than Multi-DB, but it was able to solve the same percentage of problems as Multi-DB and its search costs were lower.

We also presented results of empirical evaluations of Multi-DisPeL, comparing it first with Stoch-DisPeL and with other coarse-grained algorithms. The results show that Multi-DisPeL is quite competitive especially when the number of variables each agent holds is high. Although it is based on the same strategy with Stoch-DisPeL, agents in Multi-DisPeL are able to use the additional problem information available to them to improve search efficiency. Multi-DisPeL also fared well against Multi-AWCS, Multi-DB, and DisBO-wd in the experiments.

Chapter 8

Summary and suggestions for future work

In this final chapter, we outline the contributions of this study and summarise the key ideas and results (Section 8.1). And we also make suggestions for further work with the algorithms introduced in this thesis (Section 8.3).

8.1 Contributions

8.1.1 Landscape modification with penalties

A mechanism for modifying cost landscapes with penalties on domain values in distributed iterative improvement search was presented in Chapter 4. We argued that landscape modification with weights on constraints, which is prominent in local search, can not effectively induce search exploration in problems where the landscapes are dominated by plateaus. With an example, we showed that a finer grained approach with penalties on domain values can puncture plateaus and create new peaks in the landscape, and thus give the search more opportunities to continue its downhill descent. We also presented a mechanism for solution perturbation with penalties on domain values, introducing a temporary penalty for doing this. We showed that this has the advantage of not causing as many previously satisfied constraints to become violated; compared to perturbing a

solution with random decisions or with constraint weights.

8.1.2 Distributed Penalty Driven Search

Based on the ideas of modifying cost landscapes and perturbing solutions with penalties on domain values, we introduced the Distributed Penalty Driven Search (DisPeL) for solving DisCSPs (Chapter 5). DisPeL is a synchronous distributed iterative improvement algorithm where agents start off with a random initialisation and take turns to make sequential improvements by selecting values in the domain that minimise the sum of constraint violations and penalties. Agents resolve deadlocks with a two phased strategy, where when a deadlock is first encountered the temporary penalty is used and in repeat visits to the same deadlock incremental penalties attached to domain values are steadily increased. We also showed that while the incremental penalties are effective at modifying cost landscapes, they can also divert a search away from promising regions if they are retained for too long or are allowed to grow unbounded.

8.1.3 Stochastic Distributed Penalty Driven Search

The two phased resolution process is implemented in a deterministic fashion in DisPeL, which makes it vulnerable to the effects of bad random initialisations. To overcome this, a variation to DisPeL is presented in the form of the Stochastic Distributed Penalty Driven Search (Stoch-DisPel) (Chapter 6). In Stoch-DisPeL, agents decide randomly to use either the temporary penalty or the incremental penalty when they encounter deadlocks. We showed that with this modification, the risk of suffering from the effects of bad initialisation is minimised but it may also mean that opportunities of good initialisations may sometimes be missed.

8.1.4 Distributed Penalty Driven Search for Agents with Multiple Local Variables

Both DisPeL and Stoch-DisPeL were specifically designed for problems where each agent owned just a single variable. We noted that in realistic scenarios DisCSPs are likely to be

made of connected CSPs, where each agent represents a CSP with multiple variables. Distributed Penalty Driven Search for Agents with Multiple Local Variables (Multi-DisPeL) is introduced which extends the ideas from DisPeL and Stoch-DisPeL for the aforementioned problems (Chapter 7). In Multi-DisPeL, agents still take turns to improve a solution and they perform steepest descent search with their local variables in doing this. Agents implement penalties on their local variables and request that owners of external variables also implement the same penalties on those variables connected to the deadlocked local variables.

8.1.5 Other contributions

- **DisCSP model of the Car Sequencing Problem.** In Section 5.6.3, we describe the car sequencing problem as a DisCSP for the evaluation of DisPeL. In the model, agents represented slots on the schedule and the domain of each agent's variable was the number of different models of cars available. Global enumeration constraints for each model were introduced, to ensure that the right number of cars was placed on the schedule. To the best of our knowledge, this was the first attempt at solving this problem within the DisCSP framework.

- **Improvements to Distributed Stochastic Algorithm (DSA).** We described a modification to DSA [27, 128, 5] in Section 6.5, where we allowed only agents with inconsistent variables to make non-improving decisions to help the algorithm better deal with local optima. We found that this change (which we called DSA-B1N) improved DSA's performance by allowing it overcome the limitations highlighted in [49] and thus improving the ability to find zero cost solutions.

- **Improvements to Eisenberg's Distributed Breakout (DisBO).** DisBO [25] was modified to create DisBO-wd in Section 7.4. Dis-BO's weight update mechanism was replaced with the multiplicative and decay mechanism from [30] which allowed us to reduce the number of DisBO's different cycles from three to two, with significant cost implications. The new weight update mechanism was modified further so that weights on satisfied constraints were continuously decayed as well. We found that

DisBO-wd outperformed DisBO both in terms of the percentage of problems solved and cost of finding solutions.

8.2 Summary of results

The three new penalty driven algorithms were extensively evaluated on different problems and their performance compared with similar distributed iterative improvement algorithms. In Chapter 5, we compared DisPeL with the Distributed Breakout Algorithm (DBA) on distributed graph colouring problems, car sequencing problems, and random non-binary DisCSPs. The objective was to compare the different cost landscape modification mechanisms employed by both algorithms: penalties on domain values for DisPeL and constraint weights for DBA. Results showed that DisPeL solved more problems and the number of iterations typically required was significantly lower than that for DBA.

Stoch-DisPeL was shown to outperform DisPeL in Chapter 6, as well as our own improved version of the Distributed Stochastic Algorithm (DSA-B1N). In the experiments with random binary DisCSPs and distributed boolean satisfiability formulae, Stoch-DisPeL solved more problems within the allotted time and its search costs were lower.

Finally, the empirical evaluations in Chapter 7 show that Multi-DisPeL dominates Stoch-DisPeL in problems where each agent has a large number of variables. The evaluations in that chapter also included comparisons with coarse grained versions of other distributed iterative improvement algorithms, including Distributed Breakout and Asynchronous Weak Commitment Search. The results of the evaluation show that Multi-DisPeL is quite competitive with respect to those algorithms; in most cases its search costs are significantly lower than those incurred by the other algorithms.

In summary, all evaluations carried out have shown that although the new algorithms are incomplete, the solve a very high percentage of reasonably large DisCSPs, with up to 200 variables, within reasonable time.

8.3 Suggestions for further work

- Dynamic agent ordering: Published work on distributed backtracking algorithms has shown that dynamically changing the ordering of variables during the search can improve performance of such algorithms, by reducing the amount of redundant search carried out by agents. In our algorithms, the agent ordering determines the direction penalties flow; as such, it will be useful to study how agents ordering can be changed dynamically during the search and how such changes can be used to improve search efficiency.

- Solving Distributed Constraint Optimisation Problems (DisCOPs): We presented algorithms for solving distributed problems where the objective is find the first solution in which all constraints are satisfied. However, there are a lot of real life problems where such solutions do not exist, and the problem solving challenge is to find a solution with the least number of constraint violations or one that optimises a particular evaluation function. Extending the penalty driven algorithms for such problems would be beneficial given that they scale up quite well. However, the challenges of using these algorithms in distributed optimisation include correct termination and possibly making the algorithms complete to provide guarantees of solution optimality. Further research on extending the DisPeL algorithms for optimisation can also consider problems with soft constraints or problems where agents have preferences for particular values in their domains; in both cases the quality of valid solutions are still evaluated by defined functions that have to be optimised.

- Solving dynamic DisCSPs: In dynamic DisCSPS, the problem specification is not fixed; during a search constraints may be added or retracted, or agents may join in or drop out of the collaborative process. Key challenges include agent coordination issues and how to deal with new information without abandoning existing solutions. The DisPeL algorithms should fare well for such problems, especially since all forms of search memory held by agents are ephemeral. As such, agents are not predisposed towards the initial problem specification. Nevertheless, there are still lots of chal-

lenges in adapting our new algorithms for dynamic problems and other opportunities for improving efficiency on such problems.

- Multi-context search: The idea of solving DisCSPs with concurrent parallel searches has been around for a few years, but it has been attracting a lot of attention lately as a promising area of research. As agents take turns to act in the DisPeL algorithms, they may have a lot of idle time on their hands. This idle time can be exploited and used to carry out multiple searches; either each with different initialisations or each search being on a different ordering of agents. Either way, multiple searches will allow the algorithms to minimise the risks of having bad initialisations or unfavourable agent orderings. However, to fully benefit from parallel searches, information has to be shared between the different processes. A possibility for future work is to explore means for using information from penalties imposed on values in one search to guide other searches.
- Hybridisation and multi-algorithm search: Agents idle time can further be exploited by allowing agents use that time to run processes of another algorithm to solve the same problem. Running and sharing information with complete search algorithms can provide interesting challenges, especially where such complete algorithms are running asynchronously. For example, agents can use no-goods generated by the complete algorithms to guide the iterative improvement search. And, the complete algorithm can use information agents collect to guide their value selection. Another interesting hybrid, is a parallel run of a DisPeL algorithm with a distributed asynchronous consistency maintenance algorithm where the problem is iteratively made arc consistent with the information revealed during a search.
- There are also opportunities to explore other esoteric issues, like for example solving problems with global constraints that can not be decomposed into aggregations of smaller constraints, strengthening local inference so that agents make better decisions, and studying how the algorithms can scale up to deal with problems with thousands of variables. Further work can also consider making agents pro-active

where each agent can learn about and possibly estimate states of its neighbours during a search and with that information improve its decision making.

- Exploring our ideas in centralised settings: To the best of our knowledge, the penalty mechanism introduced in this study is entirely new and it has not been explored in centralised CSP solving. A possibility for future work is to compare how the mechanism stacks up against existing local search algorithms and if, for example, our penalty reset policy can be used to improve other such algorithms. These ideas may also be considered for problem solving on computing grids or clusters, of course with a relaxation of the DisCSP privacy assumptions.

8.4 Thesis summary

In this thesis, we investigated the idea of dealing with local optima by using penalties attached to domain values to modify cost landscapes. Our primary objective, as stated in Section 1.1, was to enhance the performance of iterative improvement algorithms for solving DisCSPs. This objective was achieved with the creation of three new algorithms (DisPeL, Stoch-DisPeL, and Multi-DisPeL) and the modifications of two existing algorithms (DSA-B1N for DSA and DisBO-wd for DisBO). The three new algorithms were based on the idea of resolving deadlocks with two types of penalties - one for perturbing a search and the other for modifying cost landscapes. We argued that modifying cost landscapes with penalties is a more effective option, as the impact on landscapes are more dramatic. This allows for quicker resumption of search exploration and as a result, improves overall search efficiency. The new algorithms were extensively evaluated on different types of problems and compared with existing distributed iterative improvement algorithms. The results reported here showed that the new algorithms were significant improvements. They solved more problems within the limited time allowed and they typically incurred significantly lower costs in the process.

Appendix A

Determining optimal parameter values for DisBO-wd

We introduced DisBO-wd a modification of DisBO [25], in Section 7.4. The weight update mechanism of DisBO is replaced with the scheme for continuous weight updates proposed in [30]. This new scheme introduces two new parameters into DisBO-wd i.e. the learning rate (lr) and the decay rate (dr). The learning rate controls how fast weights on violated constraints grow in DisBO-wd, while the decay rate biases the search towards the most recent weight increases.

In his work on SAT solving with a modified GSAT [101] algorithm, Frank [30] found that the decay rate was optimal at $dr = 0.999$, more problems were solved within an allotted time than with the value set to 0.95 and 0.99. He also found that the learning rate was optimal at $lr = 1$ compared to runs with the values 8, 16, and 24. However, DisBO-wd differs from GSAT in many respects especially given the amount concurrent changes that take place in distributed search. Therefore, we had to carry out an experiments to determine optimal values for the parameters in the distributed algorithm. We used distributed SAT instances and random DisCSPs for this experiment, evaluating DisBO's performance on 100 instances in each case, with the parameters set to $lr = [1, 2, 3, 5, 8, 10, 12, 16]$ and $dr = [0.9, 0.95, 0.98, 0.99]$. In Tables A.1 and A.2, we summarise the results from this experiment, showing the percentage of problems solved, the average and the median search

costs incurred where we limited DisBO-wd to 10,000 iterations on each attempt on the SAT problems and 12,000 iterations on each attempt on the DisCSPs.

dr	*lr*	% solved	average cost	median cost
0.9	1	60	85	28
	2	82	178	90
	3	84	151	108
	5	94	303	141
	8	88	273	139
	10	92	300	129
	12	92	233	176
	16	88	314	145
0.95	1	88	208	111
	2	98	304	144
	3	100	303	130
	5	100	329	195
	8	100	259	161
	10	98	338	181
	12	100	314	136
	16	100	358	233
0.98	1	100	236	119
	2	100	375	198
	3	100	247	174
	5	100	263	213
	8	100	286	202
	10	100	439	176
	12	100	380	219
	16	100	386	174
0.99	1	100	186	130
	2	100	252	127
	3	100	243	189
	5	100	235	139
	8	100	373	235
	10	100	312	213
	12	100	318	207
	16	100	269	213

Table A.1: Performance of DisBO-wd on Distributed SAT problems with different values for its parameters (*lr* and *dr*).

The results in Table A.1 summarise attempts to solve 50-literal SAT instances from the SATLib problem set. These results are in agreement with the findings in [30], the optimal values for DisBO-wd match those previously reported. As *dr* increases, DisBO-wd solved more problems but there is no clear relationship between the search costs and the decay

rate. With the learning rate, it is clear that the search costs increase with the value of that parameter.

dr	*lr*	% solved	average cost	median cost
0.9	1	29	1598	462
	2	79	1160	550
	3	95	2298	1458
	5	99	2412	1648
	8	100	2416	1658
	10	96	2472	1876.5
	12	93	2080	1606
	16	94	2260	1286.5
0.95	1	87	1304	663
	2	100	1986	1030.5
	3	100	2226	1414.5
	5	98	1952	1114
	8	93	1922	1238
	10	95	1876	1104
	12	96	2048	1232
	16	100	2050	1182.5
0.98	1	95	1834	1163
	2	98	1590	1050
	3	96	1568	916
	5	99	1822	1024
	8	99	2114	1038
	10	96	1476	984
	12	100	1752	1082.5
	16	98	2018	1152
0.99	1	97	1668	978
	2	97	1748	1120
	3	97	1506	832
	5	93	1528	826
	8	99	1554	858
	10	100	1682	828.5
	12	96	2152	1432.5
	16	97	2454	1376

Table A.2: Performance of DisBO-wd on 100 random DisCSPs ($< n = 60, d = 10, p1 = 0.1, p2 = 0.5 >$) with different values for its parameters (lr and dr).

The results in Table A.2 show that the parameters influence DisBO-wd differently with the random DisCSPs. Performance, in terms of search cost, again show improvements for high values of dr, this suggests that overall the search benefits from retaining some information of not too recent weight increases for as long as possible and they are not

quickly dominated by newer weight increases. However, it appears that the learning rate lr has a different effect on performance in this domain. The algorithm generally does not fare too well with the smallest and largest values for this parameter. The results, although not clear cut, show that DisBO-wd is optimal with the values 3,8, or 10 (at $dr = 0.99$), where the average search costs are minimal and the percentage of problems solved are significantly high. But, we arbitrarily chose ($lr = 8$ and $dr = 0.99$) for the experiments with the algorithm because it solved slightly more problems than with $lr = 3$ and the search costs were lower than with $lr = 10$.

Bibliography

[1] Abdulwahed M. Abbas and Edward P. K. Tsang. Constraint-based timetabling-a case study. In *2001 ACS / IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, pages 67–72. IEEE Computer Society, June 2001.

[2] Panagiotis Adamidis. Parallel evolutionary algorithms: A review. In *4th Hellenic-European Conference on Computer Mathematics and its Applications*, September 1998.

[3] 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.

[4] Aaron Armstrong and Edmund Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on ArtificialIntelligence (IJCAI)*, volume 1, pages 620–625, 1997.

[5] 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.

[6] Albert-Lszl Barabsi and Rka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[7] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, January 1994.

[8] Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161(1–2):7–24, January 2005.

[9] Christian Bessiere, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In Toby Walsh, editor, *Lecture Notes in Computer Science 2239 Springer (CP 2001)*, 2001.

[10] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996.

[11] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In Bernhard Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 309–315. Morgan Kaufmann, 2001.

[12] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, September 2003.

[13] Markus Bohlin. Constraint satisfaction by local search. Technical Report T2002:07, Swedish Institute of Computer Science, July 2002.

[14] E. Bonabeau, M. Dorigo, and G. Theraulaz. Inspiration for optimization from social insect behaviour. *Nature*, 406:39 – 42, July 2000.

[15] Ismel Brito and Pedro Meseguer. Distributed forward checking. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming (CP2003)*, volume 2833 of *LNCS*, pages 801 – 806, Berlin, September 2003. Spriger.

[16] Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSPs. In *Proceedings of the Fifth International Workshop on Distributed Constraint Reasoning*, September 2004.

[17] Tom Carchrae and J. Christopher Beck. Low-knowledge algorithm control. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 04)*, pages 49–54. AAAI Press / The MIT Press, July 2004.

[18] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on ArtificialIntelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[19] Scott Clearwater, Bernardo Huberman, and Tad Hogg. Cooperative solution of constraint satisfaction problems. *Science*, 254:1181–1183, November 1991.

[20] Phillippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhofel, editor, *Stochastic Algorithms: Foundations and Applications (SAGA 2001)*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90, Berlin, 2001. Springer.

[21] Zeev Collin, Rina Dechter, and Shmuel Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 318–324, 1991.

[22] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[23] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[24] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems - a tutorial survey. Technical report, University of California, Irvine, April 1998.

[25] Carlos Eisenberg. *Distributed Constraint Satisfaction For Coordinating And Integrating A Large-Scale, Heterogeneous Enterprise.* PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), September 2003.

[26] Cèsar Fernàndez, Ramón Béjar, Bhaskar Krishnamachari, and Carla P. Gomes. Communication and computation in distributed csp algorithms. In Pascal Van Hentenryck, editor, *8th International Conference, Principles and Practice of Constraint Programming - (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 664–679. Springer, September 2002.

[27] Stephen Fitzpatrick and Lambert Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In Kathleen Steinhofel, editor, *Lecture Notes in Computer Science 2264, 1st Syposium on Stochastoc Algorithms*, pages 49–64, Berlin, December 2001. Springer-Verlag.

[28] Stephen Fitzpatrick and Lambert Meertens. Experiments on dense graphs with a stochastic, peer-to-peer colorer. In Carla Gomes and Toby Walsh, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence AAAI-02*, pages 24–28. AAAI, AAAI Press, July 2002.

[29] Jeremy Frank. Weighting for godot: Learning heuristics for GSAT. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 96)*, volume 1, pages 338–343. AAAI Press / The MIT Press, August 1996.

[30] Jeremy Frank. Learning short-term weights for GSAT. In Martha Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 384–391, San Franciso, August 1997. Morgan Kaufmann.

[31] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[32] J. Gashnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburg, PA, 1979.

[33] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993.

[34] Fred Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989.

[35] Fred Glover. Tabu search - part II. *ORSA Journal on Computing*, 2(1):4–32, Winter 1990.

[36] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - (CP '97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, November 1997.

[37] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, February 2000.

[38] 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.

[39] Mattias Grönkvist. A constraint programming model for tail assignment. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of the First International Conference Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, volume 3011 of *Lecture Notes in Computer Science*, pages 142–156. Springer, April 2004.

[40] Youssef Hamadi. Optimal distributed arc-consistency. In Joxan Jaffar, editor, *Proceedings of the 5th Int'l Conference Principles and Practice of Constraint Programming - CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 219–233. Springer, October 1999.

[41] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools*, 11 (2):167–188, June 2002.

[42] Youssef Hamadi. Conflicting agents in distributed search. *International Journal on Artificial Intelligence Tools*, 14(3):459–476, 2005.

[43] 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.

[44] Jiming Liu. Han Jing and Cai Qingsheng. *From ALIFE Agents to a Kingdom of N Queens*, pages 110 – 120. The World Scientific Publishing Co. Pte, Ltd, November 1999.

[45] Pierre Hansen and Nenad Mladenovic. *Variable Neighbourhood Search*, pages 221–234. Oxford University Press, New York, 2002.

[46] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[47] Katsutoshi Hirayama and Makoto Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 169–177, Washington, DC, USA, April 2000. IEEE Computer Society.

[48] Katsutoshi Hirayama and Makoto Yokoo. Local search for distributed SAT with complex local problems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, AAMAS 2002, pages 1199 – 1206, New York, NY, USA, 2002. ACM Press.

[49] Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. *Artificial Intelligence*, 161(1–2):89–115, January 2005.

[50] Katsutoshi Hirayama, Makoto Yokoo, and Kaita Sycara. The phase transition in distributed constraint satisfaction problems: first results. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*, 2000.

[51] Tad Hogg. Refining the phase transition in combinatorial search. 81:127–154, March 1996.

[52] Tad Hogg and Colin P. Williams. Solving the really hard computational problems with cooperative search. In *Proceedings of AAAI93*, pages 231–236. AAAI, AAAI Press, 1993.

[53] Tag Hogg and Bernardo A. Huberman. Better than the best: The power of cooperation. In *CSSS: 1989 Lectures in Complex Systems, The Proceedings of the 1989 Complex Systems Summer School*, volume 2 of *Studies in the Sciences of Complexity*, pages 165–184, Santa Fe, NM, 1992. Addison-Wesley.

[54] Holger H. Hoos. *Stochastic Local Search - methods, models, applications.* PhD thesis, Darmstadt University of Technology, Germany, 1998.

[55] 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.

[56] 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.

[57] 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.

[58] M. D. Johnston. SPIKE: AI scheduling for hubble space telescope after 18 months of orbital operations. In *Proceedings of the 1992 AAAI Spring Symposium on Practical Approaches to Scheduling and Planning*, pages 1–5, Stanford, CA, 1992.

[59] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proc. IEEE International Conference on Neural Networks*, pages 1942 – 1948, NJ, 1995. IEEE, IEEE.

[60] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598(4598):671–680, May 1983.

[61] Kurt Konolige. Easy to be hard: Difficult problems for greedy algorithms. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 374–378. Morgan Kaufmann, May 1994.

[62] William A. Kornfeld. The use of parallelism to implement a heuristic search. In Patrick J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 575–580. William Kaufmann, august 1981.

[63] Manuel Laguna and Rafael Marti. A GRASP for coloring sparse graphs. *Computational Optimization and Applications*, 19(2):165–178, 2001.

[64] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint logic programming and option trading. *IEEE Expert*, pages 42–50, Fall 1987.

[65] Maria Lin Sui Ling. *Multi-agent Constraint Satisfaction and Optimisation.* PhD thesis, IC-Parc, Imperial College London, November 2002.

[66] J. Liu, J. Han, and Y. Y. Tang. Multi-agent constraint satisfaction. *Artificial Intelligence*, 136(1):101–144, March 2002.

[67] JyiShane Liu and Katia P. Sycara. Exploiting problem structure for distributed constraint optimization. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 246–254, San Francisco, CA, 1995. MIT Press.

[68] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[69] Arnold Maestre and Christian Bessiere. Improving asychronous backtracking for dealing with complex local problems. In Ramon López de Mántaras and Lorenza

Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI 2004)*, pages 206–210. IOS Press, August 2004.

[70] Roger Mailler and Victor Lesser. Using cooperative mediation to solve distributed constraint satisfaction problems. In *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, 2004.

[71] David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI '97)*, pages 321–326. AAAI Press / The MIT Press, 1997.

[72] Ammon Meisels and Oz Lavee. Using additional information in DisCSPs search. In Pregnesh Jay Modi, editor, *Proceedings of the 5th International Workshop on Distributed Constraint Reasoning*, September 2004.

[73] Amnon Meisels and Roie Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop (IJCAI-DCR 2003)*, 2003.

[74] Michela Milano and Andrea Roli. MAGMA: A multiagent architecture for metaheuristics. *IEEE Trans. on Systems, Men and Cybernetics – Part B*, 34(2):925–941, April 2004.

[75] Patrick Mills. *Extensions to Guided Local Search.* PhD thesis, University of Essex, Essex, 2002.

[76] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 90)*, pages 17–24. AAAI Press / The MIT Press, july 1990.

[77] Steven Minton, Mark D. Jonston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[78] Pragnesh Jay Modi. *Distributed Constraint Optimization for Multiagent Systems.* PhD thesis, University of Southern California, Department of Computer Science, 2003.

[79] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[80] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[81] Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence(AAAI 93)*, pages 40–45, 1993.

[82] T. Nguyen and Yves Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[83] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Distributed dynamic backjumping. In Pragnesh Jay Modi, editor, *Proceedings of the Fifth International Workshop on Distributed Constraint Reasoning (DCR 2004)*, September 2004.

[84] Kobbi Nissim and Roie Zivan. Secure discsp protocols - from centralised towards distributed solutions. In Amnon Meisels, editor, *Proceedings of the Sixth International Workshop on Distributed Constraint Reasoning (DCR 2005)*, pages 161–175, July 2005.

[85] Edgar M. Palmer. *Graphical evolution: an introduction to the theory of random graphs.* John Wiley & Sons, Inc., 1985.

[86] Luis Paquete and Thomas Stutzle. An experimental investigation of iterated local search for coloring graphs. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Gunther Raidl, editors, *Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002:EvoCOP, EvoIASP, EvoSTim*, pages 121–130, Kinsale, Ireland, March 2002. Springer-Verlag.

[87] Bruce D. Parrello, Waldo C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: a case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.

[88] Steven Prestwich. Random walk with continuously smoothed variable weights. In Fahiem Bacchus and Toby Walsh, editors, *Proceedings of 8th Int'l Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 203–215. Springer, June 2005.

[89] Patrick Prosser. Domain filtering can degrade intelligent backtracking search. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI '93)*, pages 262–267, Chambry, France, August 1993. Morgan Kaufmann.

[90] Patrick Prosser, Chris Conway, and Claude Muller. A constraint maintenance system for the distributed resource allocation problem. *Intell. Syst. Eng.*, 1(1):76–83, 1992.

[91] Michel Raynal. *Networks and Distributed Computation.* North Oxford Academic, London, 1987.

[92] Georg Ringwelski. Distributed and dynamic constraint processing in two agent layers. Technical report, Cork Constraint Computation Centre, Universtity of Cork, Cork Ireland, October 2003.

[93] Georg Ringwelski. The DDAC4 algorithm for arc-consistency enforcement in dynamic and distributed discsp. In *Proceedings of the Fifth International Workshop on Distributed Constraint Reasoning*, September 2004.

[94] Georg Ringwelski. Incremental constraint propagation for interleaved distributed backtracking. In *Proceedings of the Distributed Constraints Reasoning Workshop (CP-DCR 2004)*, September 2004.

[95] Georg Ringwelski and Youssef Hamadi. Boosting distributed constraint satisfaction. In Peter van Beek, editor, *Proceedings of the 11th International Conference Prin-*

ciples and Practice of Constraint Programming - CP 2005, volume 3709 of *Lecture Notes in Computer Science*, pages 549–562. Sringer, September 2005.

[96] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, pages 10–20. Springer, 1994.

[97] Arvind Sathi and Mark Fox. Constraint-direction negotiation of resource reallocations. Technical Report CMU-RI-TR-89-12, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 1989.

[98] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, editor, *Proceedings International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 631–639, Montreal, August 1995. Morgan Kaufmann.

[99] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, pages 297–302. AAAI Press / The MIT Press, July 2000.

[100] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, (IJCAI 2001)*, pages 334–341. Morgan Kaufmann, August 2001.

[101] Bart Selman and Henry A. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI '93)*, pages 290–295, August 1993.

[102] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94)*, volume 1, pages 337–343. AAAI Press, July 1994.

[103] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 92)*, pages 440–446. The AAAI Press / The MIT Press, July 1992.

[104] Marius-Calin Silaghi, Amit Abhyankar, Markus Zanker, and Roman Barták. Deskmates (stable matching) with privacy of preferences, and a new distributed csp framework. In Ingrid Russell and Zdravko Markov, editors, *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2005)*, pages 671–677. AAAI Press, May 2005.

[105] Marius-Calin Silaghi and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161(1-2):25–53, January 2005.

[106] Marius-Calin Silaghi, D. Sam-Haroud, and Boi Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report 364, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), May 2001.

[107] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, pages 917–922, Austin, Texas, August 2000. AAAI Press.

[108] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. ABT with asynchronous reordering. In *2nd Asia-Pacific Conference on Intelligent Agent Technology (IAT'2001)*, pages 54–63. World Scientific, 2001.

[109] Christine Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions On Evolutionary Computation*, 6(4):347– 357, August 2002.

[110] Gadi Solotorevsky, Ehud Gudes, and Amnon Meisels. Distributed constraint satisfaction problems - a model and application. 1997.

[111] Dave Tompkins and Holger Hoos. Warped landscapes and random acts of SAT solving. In *8th International Symposium on Artificial Intelligence and Mathematics (AMAI 2004)*, January 2004.

[112] Michel Toulouse, Teodor Crainic, and Brunilde Sanso. Systemic behavior of cooperative search algorithms. *Parallel Computing*, 30(1):57–79, January 2004.

[113] Christos Voudouris. *Guided local search for combinatorial optimisation problems.* PhD thesis, University of Essex, Colchester, UK, July 1997.

[114] Christos Voudouris. Guided local search: An illustrative example in function optimisation. *BT Technology Journal*, 16(3):46–50, July 1998.

[115] Christos Voudouris and Edward Tsang. Solving the radio link frequency assignment problem using guided local search. In *Proceedings of the NATO Symposium on Radio Length Frequency Assignment*, October 1998.

[116] Benjamin W. Wah and Zhe Wu. The theory of discrete lagrange multipliers for nonlinear discrete optimization. In *Principles and Practice of Constraint Programming*, pages 28–42, 1999.

[117] 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.

[118] David Waltz. Understanding line drawings of scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, New York, 1975.

[119] Lars Wittenburg. Distributed constraint solving and optimizing for micro-electromechanical systems. Master's thesis, Technical University of Berlin, Decebmer 2002.

[120] Xiaolong Jin Yi Tang, Jiming Liu. Adaptive compromises in distributed problem solving. In *Proceedings of the 4th International Conference on Intelligent Data Engineeringand Automated Learning IDEAL 2003*, March 2003.

[121] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In Ugo Montanari and Francesca Rossi, editors, *Proceeidings of the First International Conference Principles and Practice of Constraint Programming (CP '95)*, volume 976 of *Lecture Notes in Computer Science*, pages 88–102. Springer, September 1995.

[122] 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.

[123] 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.

[124] Makoto Yokoo and Katsutoshi Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, pages 372–379, Washington, DC, USA, July 1998. IEEE Computer Society.

[125] Makoto Yokoo, Koutarou Suzuki, and Katsutoshi Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In Pascal Van Hentenryck, editor, *Proceedings of the 8th International Conference Principles and Practice of Constraint Programming - (CP '02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 387–401. Springer, September 2002.

[126] Z.Habbas, F. Herrmann, P.-P. Mrel, and D. Singer. Parallel search algorithms for constraint satisfaction problems. In Dominique de Werra and Thomas M. Liebling, editors, *Proceedings of the 16th International Symposium on Mathematical Programming*, Lausanne, July 1997.

[127] Weixiong Zhang and Lars Wittenburg. Distributed breakout revisited. In *Proceedings*

of the Eighteenth National Conference on Artificial Intelligence, pages 352 – 357. AAAI Press, July 2002.

[128] 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.

[129] Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 316–321. Morgan Kaufmann, August 2001.

[130] Lingzhong Zhou, John Thornton, and Abdul Sattar. Dynamic agent-ordering and nogood-repairing in distributed constraint satisfaction problems. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the 17th Florida Artificial Intelligence Research Symposium Conference (FLAIRS)*, Maimi, May 2004. AAAI Press.

[131] Roie Zivan and Amnon Meisels. Concurrent backtrack search on DisCSPs. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the 17th International Florida Artificial Intelligence Research Symposium Conference*. AAAI Press, 2004.

[132] Roie Zivan and Amnon Meisels. Dynamic reordering for asynchronous backtracking on DisCSPs. In Amnon Meisels, editor, *Proceedings of the Sixth International Workshop on Distributed Constraint Reasoning (DCR 05)*, pages 15–29, July 2005.