ROBERT GORDON UNIVERSITY ABERDEEN

Combining Search Strategies for Distributed Constraint Satisfaction

Amina Sambo-Muhammad Magaji

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

August 2015

Abstract

Many real-life problems such as distributed meeting scheduling, mobile frequency allocation and resource allocation can be solved using multi-agent paradigms. Distributed constraint satisfaction problems (DisCSPs) is a framework for describing such problems in terms of related subproblems, called a complex local problem (CLP), which are dispersed over a number of locations, each with its own constraints on the values their variables can take. An agent knows the variables in its CLP plus the variables (and their current value) which are directly related to one of its own variables and the constraints relating them. It knows little about the rest of the problem. Thus, each CLP is solved by an agent which cooperates with other agents to solve the overall problem.

Algorithms for solving DisCSPs can be classified as either systematic or local search with the former being complete and the latter incomplete. The algorithms generally assume that each agent has only one variable as they can solve DisCSP with CLPs using "virtual" agents. However, in large DisCSPs where it is appropriate to trade completeness off against timeliness, systematic search algorithms can be expensive when compared to local search algorithms which generally converge quicker to a solution (if a solution is found) when compared to systematic algorithms. A major drawback of local search algorithms is getting stuck at local optima. Significant researches have focused on heuristics which can be used in an attempt to either escape or avoid local optima.

This thesis makes significant contributions to local search algorithms for DisCSPs. Firstly, we present a novel combination of heuristics in DynAPP (Dynamic Agent Prioritisation with Penalties), which is a distributed synchronous local search algorithm for solving DisCSPs having one variable per agent. DynAPP combines penalties on values and dynamic agent prioritisation heuristics to escape local optima. Secondly, we develop a divide and conquer approach that handles DisCSP with CLPs by exploiting the structure of the problem. The divide and conquer approach prioritises the finding of variable instantiations which satisfy the constraints between agents which are often more expensive to satisfy when compared to constraints within an agent. The approach also exploits concurrency and combines the following search strategies: (i) both systematic and local searches; (ii) both centralised and distributed searches; and (iii) a modified compilation strategy. We also present an algorithm that implements the divide and conquer approach in Multi-DCA (Divide and Conquer Algorithm for Agents with CLPs).

DynAPP and Multi-DCA were evaluated on several benchmark problems and compared to the leading algorithms for DisCSPs and DisCSPs with CLPs respectively. The results show that at the region of difficult problems, combining search heuristics and exploiting problem structure in distributed constraint satisfaction achieve significant benefits (i.e. generally used less computational time and communication costs) over existing competing methods.

Keywords: Distributed constraint satisfaction, Distributed problem solving, local search, algorithms, local optima, heuristics.

Acknowledgments

To Almighty God be the glory. I would like to appreciate my supervisors Ines Arana and Hatem Ahriz for being outstanding mentors. Their insightful guidance and invaluable feedbacks contributed immensely to the completion of this thesis. My sincere gratitude goes to the entire staff of the school of computing and digital media for providing me with an enabling atmosphere. A special thank you also goes to the ITS staff for their support in providing access to systems for my numerous experiments.

I would like to extend my gratitude to Mama and Muhammad for being my pillar of strength and for their unwavering love and prayers. I wish you were here dad to see one of your visions manifest to reality. To my babies Aisha, Zahra, Muhammad jnr and Ibrahim thank you for tolerating me through my moments. Finally, I would like to acknowledge my extended family members, friends, colleagues, and acquaintances for their many kind and encouraging words.

This thesis was fully funded by a research studentship from the Petroleum Technology Development Fund, Nigeria.

Declarations

I hereby declare that I am the sole author of this thesis. All other work and sources of information contained in this thesis have been diligently cited in the bibliography.

Parts of the work presented in this thesis have appeared in the following publications.

CHAPTER 5

A. Sambo-Magaji, I. Arana, and H. Ahriz, Dynamic Agent Prioritisation with Penalties in Distributed Local Search, in Proceedings of The 5th International Conference on Agents and Artificial Intelligence (ICAART 2013), Volume 1, pp. 317-322, 15th-18th February 2014, Spain, Barcelona.

CHAPTER 6

A. Sambo-Magaji, I. Arana, and H. Ahriz, Local Search Algorithm for DisCSPs with Complex Local Problems, in Proceedings of 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2014), pp. 56-63, 11th-14th August 2014, Warsaw, Poland.

Contents

List of Figures

List of Tables

Abbreviations

AWCS	Asynchronous Weak Commitment Search
CSP, CSPs	Constraint Satisfaction Problem(s)
costwTB	Stochastic Distributed Penalty Driven Search with Weights for tie breaking cost
DBA	Distributed Breakout Algorithm
DBHyb	Weight-Based Distributed Hybrid Algorithm
DCOP, DCOPs	Distributed Constraint Optimization Problem(s)
DisBO-wd	Distributed Breakout with Weight Decay
DisCSP, DisCSPs	Distributed Constraint Satisfaction Problem(s)
DisHyb	Distributed Knowledge-Based Hybrid Approach
DisPeL	Distributed Penalty Driven Search
Dpenalties	Single Distributed Breakout with Weight Decay and Penalties
DynAPP	Dynamic Agent Priority with Penalties
DynCW	Dynamic Agent Prioritisation with Constraint Weights
Multi-DB	Distributed Breakout Algorithm for Agents with Multiple Local Variables
Multi-DCA	Divide and Conquer Algorithm for Agents with Complex Local Problems
Multi-DisPeL	Distributed Penalty Driven Search with Multiple Local Variables
Multi-DynAPP	Dynamic Agent Prioritisation with Penalties for Agents with Complex Local Problems
Multi-HDCS	Hybrid Distributed Concurrent Search Framework for Agents with Multiple Local Variables
Multi-HDCS-DB	Weight-Based Hybrid Distributed Concurrent Search Framework for Agents with Multiple Local Variables

Multi-HDCS-Pen	Penalty-Based Hybrid Distributed Concurrent Search Framework for Agents with Multiple Local Variables
Multi-Hyb-DB	Weight-Based Hybrid Algorithm for Agents with Multiple Local Variables
Multi-Hyb-Pen	Penalty-Based Hybrid Algorithm for Agents with Multiple Local Variables
NCCC(s)	Non Concurrent Constraint Check(s)
normSweights	Normalised Stochastic Distributed Penalty Driven Search with Weights
PenDHyb	Penalty-Based Distributed Hybrid Algorithm
SingleDB	Single Distributed Breakout
SingleDB-wd	Single Distributed Breakout with Weight Decay
SingleDB-wd+Restart	Single Distributed Breakout with Weight Decay and periodic restart
Stoch-DisPeL	Stochastic Distributed Penalty Driven Search
StochTB	Stochastic Distributed Penalty Driven Search with Tie Breaking
StochTBP	Stochastic Distributed Penalty Driven Search with Probabilistic Tie Breaking
StochDisPeL+Restart	Stochastic Distributed Penalty Driven Search with periodic restart
Sweight	Stochastic Distributed Penalty Driven Search with Weight

Chapter 1

Introduction

Distributed Constraint Satisfaction Problem (DisCSP) is a framework which extends constraint satisfaction in order to address problem distribution. Informally, a **Constraint Satisfaction Problem** (CSP) is a paradigm for representing a problem to consist of decision **variables**, the **domain** of each variable and **constraints** between variables that must be satisfied. However, some CSPs are naturally distributed, i.e. the problem is not centralised, but divided into a number of subproblems which are dispersed over a number of locations. In solving such problems, a centralised problem solving approach may not be possible due to cost and privacy restrictions. Thus, the centralised constraint satisfaction framework becomes insufficient and these problems lend themselves to solutions using distributed constraint satisfaction.

Simplistically, a DisCSP is a CSP which can be divided into inter-related subproblems (smaller CSPs) each of which is assigned to an agent which is responsible for solving it. Thus, each agent has its own variables, domains and constraints (CSP) as well as sharing some constraints with agents whose CSP its variables are related to. Agents are responsible for finding suitable instantiations for the variables they represent. Within each agent, the variables together with their domains and constraints form a cluster known as a **Complex Local Problem** (CLP). Each agent knows about its own complex local problem but knows little about the rest of the problem (other agents' complex local problems) - it only knows the variables (and current values) which are directly related to one of its own variables together with the constraints relating them. The objective of a DisCSP algorithm is to assign values to all variables that satisfy all the constraints associated with the problem. To satisfy all constraints, each agent has to solve its own CLP and also needs to negotiate with other agents to ensure that the solution to the CLPs can be aggregated into a satisfactory solution to the overall problem. For example, in a distributed meeting scheduling scenario, each department (agent) may have several meetings (variables) to schedule by assigning it a timeslot (domain). Some meetings may involve only participants within a department (i.e. within its complex local problem) while others involve participants from other departments (i.e. CLPs in other agents). To find a solution to the overall meeting scheduling problem, each agent must schedule the meetings in its CLP and negotiate with other agents to schedule meetings involving participants in other agents.

DisCSPs algorithms generally assume a single variable per agent and that this can be easily extended to solve DisCSP with CLPs using "virtual" agents. The algorithms can be broadly classified as either systematic or local search. In large DisCSPs where it is appropriate to trade completeness off against timeliness, local search algorithms are more effective and perform favourably by converging quicker to a solution (if a solution is found) when compared to systematic search algorithms. Local search algorithms however have a major drawback of getting stuck at local optima and several heuristics have been used to deal with local optima. A challenge in implementing local search algorithms for DisCSP remains in designing algorithms that improve search by escaping local optima more effectively.

1.1 Research Objective

This thesis aims to make significant contributions in local search algorithms for solving DisCSPs by proposing a novel approach to problem solving which combines existing heuristics while exploiting the structure of DisCSPs. Our main research question is *"Are there any benefits in combining existing strategies for distributed local search?"*. This question has resulted in the following research objectives:

1. Investigate the effect of combining existing search heuristics to escape local optima in local search algorithms for distributed constraint satisfaction problems.
2. Exploit the structure of distributed constraint satisfaction problems with complex local problems to develop a more efficient approach to solving them.

1.2 Key Contributions of the Thesis

This thesis contributes to the development of algorithms for DisCSPs. We combined several heuristics in distributed local search and also exploited the structure of DisCSPs with complex local problems to propose new solving strategies. The primary contributions of this thesis are summarised next.

First: The first contribution of this thesis is **DynAPP** - Dynamic Agent Prioritisation with Penalties. DynAPP is an algorithm for one variable per agent that combines two heuristics: (i) dynamic agent prioritisation; and (ii) penalties on variable values. *Penalties on values* is a fine-grained heuristic that avoids values which are often found at local optima, while *dynamic agent prioritisation* changes the priority of finding a consistent value for the agent's variable.

Second: The second contribution is the **Divide and Conquer Approach** for the resolution of distributed constraint satisfaction problems with complex local problems. The approach prioritises the satisfaction of the distributed part of the problem (i.e. the inter-agent part) which is more expensive to satisfy compared to an agent's CLP. The divide and conquer approach concurrently uses: (i) several systematic searches and a local search; (ii) several centralised searches and a distributed search; and (iii) a modified compilation reformulation strategy. In solving a DisCSP with CLPs, not all variables are considered in the first instance. The CLP of each agent is considered to identify smaller clusters that may exist to form *compound groups*. The divided problem is then solved concurrently with three different types of searches as follows: (i) each agent carries out a centralised systematic search for each of its *compound groups*; (ii) a single distributed local search algorithm combines the solutions to the *compound groups* to satisfy the distributed constraints (relating the CLPs); and (iii) each agent performs one systematic search locally to extend its selected compound group solutions to a complete solution. Hence, the divide and conquer approach interleaves several searches, thus, exploits concurrency. We also implemented **Multi-DCA** - an overall local search algorithm that implements the divide and conquer approach for solving DisCSP with CLPs.

1.3 Scope of Study

The main focus of this thesis is on Distributed Constraint Satisfaction Problems (DisCSPs). The scope of this study is established by the following decisions and assumptions.

1. Satisfaction: the first set of assignments that satisfies all the constraints in the DisCSP is returned although several solutions may exist.

2. Constraints: We consider binary constraints (i.e. between two variables); this is an assumption

which is often made (Yokoo & Hirayama 2000) as a CSP with constraints that are non-binary can be converted to have only binary constraints (Bacchus & Run 1995). Although not normally counted by researchers, these transformations can be costly (Bessière 1999).

3. Communication: Communication is between neighbour agents i.e. agents whose CLPs are related by one or more constraints.

4. The underlying communication network is reliable: Thus, there is a finite message delay with messages between two agents. Messages arrive in the order in which they are sent (Yokoo, Durfee, Ishida & Kuwabara 1998).

1.4 Thesis Structure

This thesis is divided into seven chapters and three appendices. It is organized as follows:

Chapter 2 **Distributed Constraint Problems:** We define and explain the Constraint Satisfaction Problem (CSP) and Distributed Constraint Satisfaction Problem (DisCSP). We also describe Constraint Optimisation Problem (COP) and Distributed Constraint Optimisation Problem (DCOP), a specialization of DisCSP concerned with valued constraints. Privacy in distributed constraint problems and some terminologies used in the thesis are also presented.

Chapter 3 **Benchmark Problems:** we discuss four DisCSP classes used to empirically evaluate our algorithms which are: distributed graph colouring problems, random distributed constraint satisfaction problems, distributed meeting scheduling problems and distributed sensor networks problems. For each problem class, we explain how they are formulated as a distributed constraint satisfaction problem. The performance metrics used for the empirical evaluation of our algorithms are also discussed.

Chapter 4 **Algorithms for solving Distributed Constraint Problems:** Existing distributed constraint problem solving algorithm use search methods that are generally categorized as: (i) synchronous or asynchronous; and (ii) local search or systematic search methods. We present a literature review of existing algorithms for distributed constraint satisfaction problems for single variable per agent and those where an agent represents several variables. We also review and critically analyse algorithms for distributed constraint optimisation problems which are a specialisation of distributed constraint satisfaction problems.

Chapter 5 Combination Heuristics in Local Search for DisCSPs with One Variable/Agent: The concept of combining heuristics in distributed local search is investigated in this chapter by combining constraint weights, value penalties and dynamic agent prioritisation. Several algorithms for solving DisCSPs with a single variable per agent were implemented with the combination heuristics. DynAPP (Dynamic Agent Prioritisation with Penalties) is an algorithm that combines dynamic agent prioritisation and value penalties. DynAPP was found to perform best overall and is discussed in details in this chapter. Other combinations of heuristics are constraint weights and value penalties which is presented in Appendix A; and multi-context search in, Appendix B.

Chapter 6 Exploiting Structure in DisCSPs with Complex Local Problems: In this chapter, the structure of DisCSPs with CLPs is exploited. We present a novel divide and conquer approach for handling complex local problems in DisCSPs that takes into account the problem structure. The divide and conquer approach divides and solve a DisCSP by combining the following search strategies: (i) systematic and local searches; (ii) centralised and distributed searches; and (iii) a modified compilation reformulation strategy. The divide and conquer approach is implemented in Multi-DCA, an overall local search algorithm for solving DisCSPs with CLPs. Multi-DCA is discussed and empirically evaluated.

Chapter 7 Future Work and Conclusion: Finally, we conclude with a summary of the contributions of the thesis and we propose some future work.

Chapter 2

Distributed Constraint Problems

2.1 Introduction

Distributed Constraint Satisfaction Problem (DisCSP) is a formalism for modelling many distributed problems. The objective (solution) of a DisCSP is to find a complete assignment that **satisfies** all constraints in the problem. Distributed Constraint Optimization Problem (DCOP) extends DisCSP for finding the **best approximation** to a solution (a solution may not exist) that optimises a given objective function. In this chapter, we define DisCSPs, DisCSPs with CLPs and DCOPs. We also discuss privacy in distributed problems and explain some related terms.

This chapter is organized as follows. In Section 2.2 we present the formalism of a DisCSP. This is followed by the formalism of a DCOP in Section 2.3. Privacy in distributed problems is discussed in Section 2.4 and some related terminologies are described in Section 2.5. Finally, the chapter is summarised in Section 2.6.

2.2 Distributed Constraint Satisfaction

2.2.1 Background: Constraint Satisfaction Problem (CSP)

A CSP (Dechter 2003) is a problem defined as a tuple comprising of {X, D, C} such that: X = $\{x_1$, ..., $x_n\}$ represents a finite set of *variables*; D = $\{D_{x_1}$, ..., $D_{x_n}\}$ represents a set of discrete, finite *domains*, one per variable x_i, i $\in$ {1, ..., n}; and C = $\{c_1$, ..., $c_k\}$ represents a finite set of *constraints* between variables. A

solution to a CSP is a complete assignment of variables with values from their domains which satisfies all constraints.

2.2.2 Distributed Constraint Satisfaction Problem (DisCSP)

A DisCSP (Yokoo et al. 1998) is a CSP which is dispersed over a number of agents in different locations. Thus, agents have to cooperate to solve the overall problem. Each agent in a DisCSP is responsible for the assignment of values to its own variable(s) and must communicate with other agents that it shares a constraint with, about its current assignments. Thus, the main objective (**solution**) of a DisCSP is to assign values to variables that satisfy all the constraints. Formally, a DisCSP comprises of four components which are represented as a tuple $\{$A, X, D, C$\}$ where:

- A = $\{a_1$, ..., $a_m\}$ represents a set of *agents*;
- X = $\{x_1$, ..., $x_n\}$ represents a finite set of *variables*, where each variable is assigned to a single agent;
- D = $\{D_{x_1}$, ..., $D_{x_n}\}$ represents a set of finite, discrete *domains*, one per variable x_i, i $\in$ $\{1, ..., n\}$; and
- C = $\{c_1$, ..., $c_k\}$ represents a finite set of *constraints* between variables.

A constraint graph is used to illustrate a DisCSP where nodes are variables and links represent the constraints. Figure 2.1 is a simple example of a DisCSP with an agent representing a single variable. The problem involves the allocation of timeslots for 5 student presentations. Modelling this as a DisCSP, the students are the *variables* X = $\{$a, b, c, d, e$\}$ and are represented by a letter inside a circle. Agent A, Agent B, Agent C, Agent D and Agent E each represents a single variable a, b, c, d, e respectively. D$_{x_i}$ where x_i $\in$ $\{$a, b, c, d, e$\}$ represents the *domain* (timeslots) that can be assigned to each student. The domain for each variable is D_a = $\{1, 2\}$, D_b = $\{1, 2\}$, D_c = $\{1, 3\}$, D_d = $\{1, 3, 4\}$, D_e = $\{2, 3\}$. Constraints between the variables are illustrated with a line between variables. The number of constraint violations is depicted by Viol$_{x_i}$ and determined for each value in a domain of a variable while keeping the current value of its neighbours. Constraints in red are violated. The *constraints* are: (i) (a = b), i.e. a and b must be assigned the same slot; (ii) (b $\neq$ c), i.e. b and c cannot be assigned the same slot; (iii) (b = e), i.e. b and e must be assigned the same slot; (iv) (c $\neq$ d), i.e. c cannot be assigned the same slot as d; and (v) (c $>$ e), i.e. c must be assigned a time slot after e.

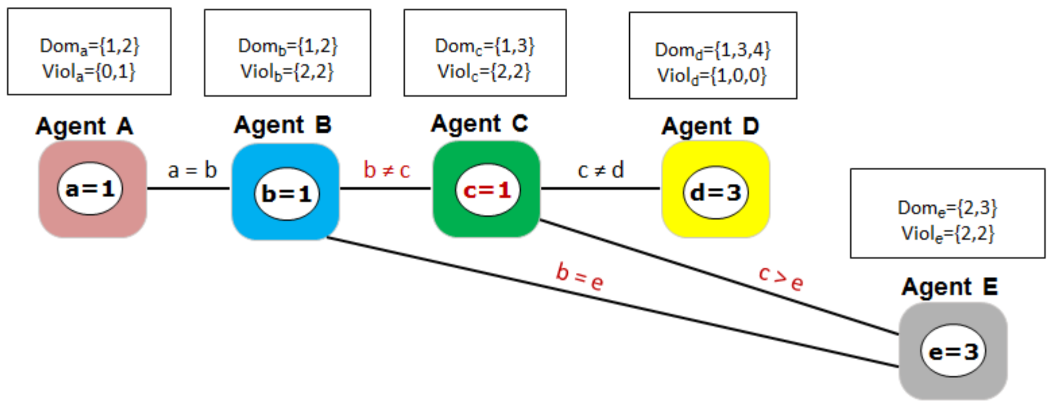


Figure 2.1: An example of a DisCSP with one variable/agent

A general assumption in algorithms for distributed constraint satisfaction is that an agent represents a single variable and the algorithms can be extended using “virtual” agents to handle DisCSP with multiple local variables per agent (Yokoo & Hirayama 2000). However, multiple local variables form a **Complex Local Problem (CLP)**, thus, a variable has constraints with variables in its CLP and may also share a constraints with variables in other CLPs. CLPs introduce additional opportunities to speed-up resolution as each agent knows more about the problem. Next we discuss DisCSP with CLPs.

2.2.3 DisCSP with Complex Local Problems

Also known as coarse grained DisCSP, a DisCSP with Complex Local Problems (CLPs) is a DisCSP framework where an agent may represent two or more variables (Yokoo 1995a). Thus, two types of constraints are considered. Constraints between variables represented by the same agent which are referred to as **intra-agent constraints** and constraints between variables that are represented by different agents called **inter-agent constraints**. Each agent holds information on its own complex local problem (current values, variables, domain values and intra-agent constraints). An agent also knows the inter-agent constraints its variables have with other variables belonging to other agents (together with their current assignments). Consequently, agents are faced with finding solutions that satisfy both intra-agent constraints as well as the inter-agent constraints their variables are involved in.

For example, Figure 2.2 describes 4 agents (departments) (Agent A, Agent B, Agent C, Agent D) in a meeting scheduling problem (discussed in Chapter 3, Section 3.4) involving 55 variables (meetings). Each agent has several variables to instantiate. Each variable is represented by a number inside a circle, the black lines between variables represent inter-agent constraints and the green lines represent intra-agent

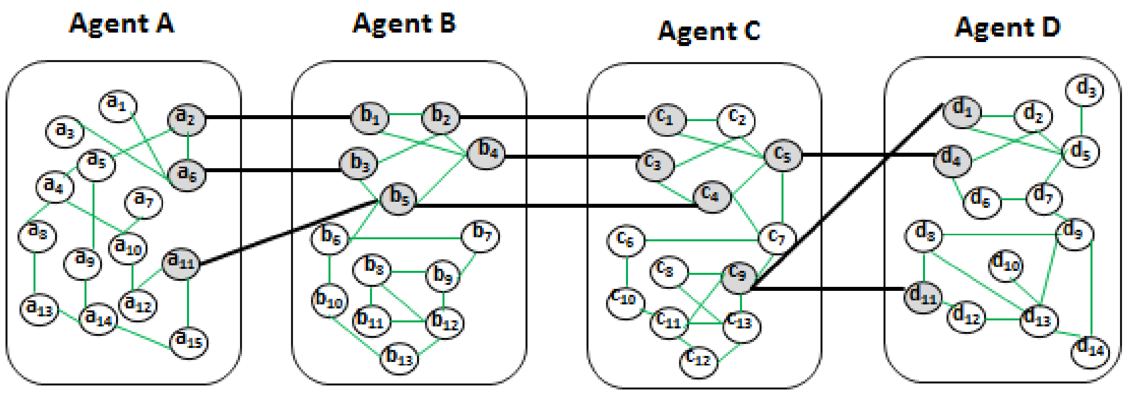


Figure 2.2: An example of a DisCSP with CLPs

constraints. Next, with this example, we explain some related terms.

Number of variables per agent: This defines the number of variables that an agent represents i.e the number of variables in each CLP. Each agent may have a different number of variables. For example, in Figure 2.2, Agent A has 15 variables $\{$ a_1, ..., $a_{15}\}$.

Intra-agent constraints: These are the constraints between variables belonging to the same agent. In Figure 2.2, these are represented by the green lines connecting variables. For example, Agent A has 16 intra-agent constraints between variables $\{(a_1, a_6), (a_2, a_5), (a_2, a_6), (a_3, a_6), (a_4, a_5), (a_4, a_8), (a_4, a_{10}), (a_5, a_9), (a_7, a_{10}), (a_8, a_{13}), (a_9, a_{14}), (a_{10}, a_{12}), (a_{11}, a_{12}), (a_{11}, a_{15}), (a_{13}, a_{14}), (a_{14}, a_{15})\}$.

Inter-agent constraints: These are the constraints a variable has with other variables that belong to different agents. In Figure 2.2, these are represented by the black lines connecting variables. For example, Agent A has 3 inter-agent constraint between variables $\{(a_2, b_1), (a_6, b_3), (a_{11}, b_5)\}$.

External variables: These are the variables which are involved in at least one inter-agent constraint. External variables are also involved in intra-agent constraints. In Figure 2.2, these are enclosed in a grey circle. For example, Agent A has 3 external variables $\{a_2, a_6, a_{11}\}$.

Internal variables: These are the variables that are involved in only intra-agent constraints. In Figure 2.2, these are enclosed in a white circle. For example, Agent A has 12 internal variables $\{a_1, a_3, a_4 ,a_5, a_7, a_8, a_9, a_{10}, a_{12}, a_{13}, a_{14}, a_{15}\}$.

Proportion of internal (external) variables: This is the ratio of the internal to external variables from the total number of variables in the problem. From the example in Figure 2.2, Agent A has 15 variables from which 12 are internal variables which represents 80% to give an 80(20) internal(external) variables.

Proportion of intra (inter) constraints: This is the ratio of the intra (inter) constraints from the total number of constraints in the problem. From the example in Figure 2.2, variables in Agent A have a total of 19 constraints, from which 16 are intra-agent constraints which represents 84% to give an 84(16) intra (inter) constraints.

Reformulation Strategies for DisCSP with CLPs

Compilation and decomposition reformulation strategies have been proposed (Yokoo & Hirayama 2000) in order to deal with DisCSPs with CLPs and implemented in several algorithms (Yokoo 1995a, Armstrong & Durfee 1997, Hirayama & Yokoo 2002, Maestre & Bessière 2004, Mueller & Havens 2005, Burke 2008). In **compilation**, agents take advantage of the centralized nature of the CLP by finding complete solutions to each agent's CLP and these solutions are combined with solutions from other agents to form a complete solution. Using this reformulation on DisCSPs with a large number of variables and constraints in each CLP, can be expensive and can result in wasteful search of areas that do not belong in the distributed solution.

For example, to reformulate the problem in Figure 2.2 using compilation (see Figure 2.3), Agent A finds local solutions to its variables $\{a_1, ..., a_{15}\}$, Agent B finds local solutions to its variables $\{b_1, ..., b_{13}\}$, Agent C finds local solutions to its variables $\{c_1, ..., c_{12}\}$, Agent D finds local solutions to its variables $\{d_1, ..., d_{14}\}$ that satisfy intra-agent constraints (illustrated by green lines). Thus, the problem is reformulated to a DisCSP with 4 external variables {Avars, Bvars, Cvars, Dvars}, each having a domain of compound values. For example, if each variable in Agent A has a domain {0, ..., 7}, the possible domain of Agent A would be {(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0), (0,0,0,0,0,0,0,0,0,0,0,0,0,0,1), ..., (7,7,7,7,7,7,7,7,7,7,7,7,7,7,7)}, a solution to agent A's CLP if it satisfies all the intra-agent constraints. The domain of Avars is the set of value combinations which satisfy all the intra-agent constraints for agent As CLP. The values in the domain of each agents compound variable are then combined to form a solution that satisfies the inter-agent constraints.

However, with **decomposition**, virtual agents are created for each individual variable, thus simplifying the problem to one of a single variable per agent. Consequently, intra-agent constraints are treated as inter-agent constraints. An agent does not take advantage of the centralized problem within a CLP but rather

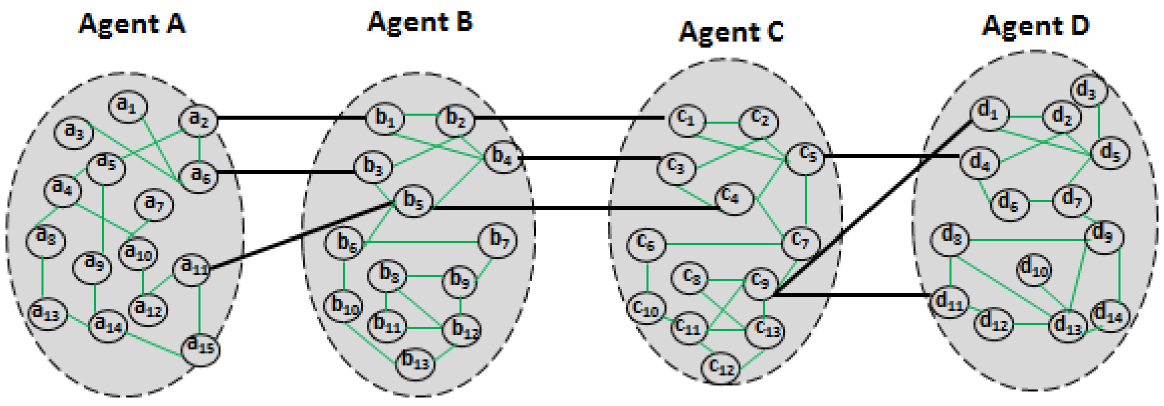


Figure 2.3: Compilation of a DisCSP with CLPs

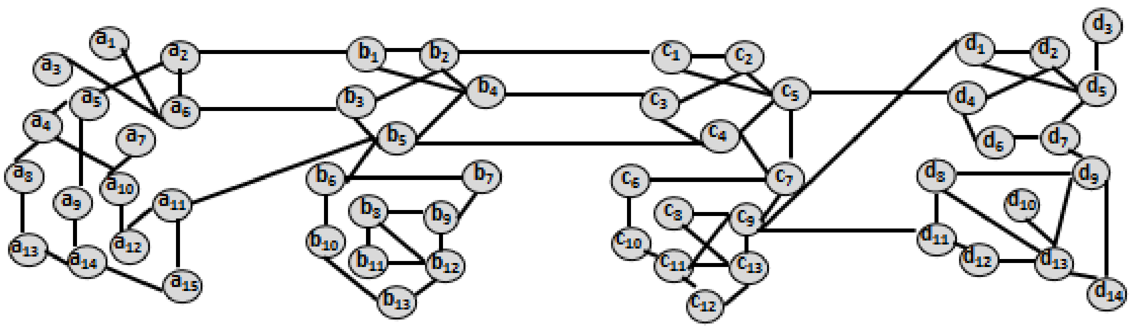


Figure 2.4: Decomposition of a DisCSP with CLPs

solves the whole problem distributively.

With reference to the example in Figure 2.2, the problem is reformulated to a DisCSP with 55 external variables and 87 inter-agent constraints. Each of the variables (a_1, ..., a_{15}, b_1, ..., b_{13}, c_1, ..., c_{12}, d_1, ..., d_{14}) is represented by a virtual agent, thus, it is treated as an external variable (see Figure 2.4). A comprehensive comparative analysis and trade-off of both approaches is found in (Burke & Brown 2006a).

2.3 Distributed Constraint Optimization

Constraint Optimisation Problem (COP) A COP (Schiex, Fargier & Verfaillie 1995) is an extension of a CSP to address optimization problems. More formally, a COP is a tuple (X, D, R) such that:

X = $\{$x$_1$, ..., x$_n\}$ represents a set of *variables*; D = $\{D_{x_1}$, ..., $D_{x_n}\}$ represents a set of discrete, finite variable *domains*; and C = $\{c_1$, ..., $c_k\}$ represents a set of *constraints* between variables where each constraint is a function that maps every possible variable assignment to a cost. Thus, the **solution** (objective) to a COP is to find an assignment for all variables that optimizes a given objective function (i.e. minimise the aggregated sum of constraint costs for the given assignment of variables).

Distributed Constraint Optimization Problem (DCOP) A natural extension to DisCSPs are Distributed Constraints Optimization Problems (**DCOPs**), (Hirayama & Yokoo 1997), (Mailler & Lesser 2004), (Modi, Shen, Tambe & Yokoo 2005), (Petcu & Faltings 2005). A DCOP consists of:

- A = $\{$a$_1$, ..., a$_m\}$ represents a set of agents;
- X = $\{$x$_1$, ..., x$_n\}$ represents a set of *variables*, where each variable is assigned to a single agent;
- D = $\{D_{x_1}$, ..., $D_{x_n}\}$ represents a set of finite, discrete variables *domains*; and
- C = $\{c_1$, ..., $c_k\}$ represents a set of constraints i.e. cost functions defined over variables.

The objective (**solution**) to a DCOP is for an agent to find an assignment for all its variables that minimise the sum of the cost functions for a given assignment of variables.

Max-DisCSP (Modi et al. 2005), Multiply-Constrained DCOP (**MC-DCOP**) (Bowring & Tambe 2006), Multi-Variable Multiply-Constrained DCOP (**MV-MC-DCOP**) (Portway & Durfee 2010) and Asymmetric DCOP (**ADCOP**) (Grinshpoun, Grubshtein, Zivan, Netzer & Meisels 2013) are extensions of DCOP that address some specified problem features.

2.4 Privacy in Distributed Constraint Problems

While in a centralised constraint problem, the information about all the variables, domain values and constraints is known centrally, in a distributed constraint problem, full information is not available at any location, i.e. each agent (location) only knows its complex local problem together with the “current” value assignments for variables connected to its own variables via constraints. Hence, the additional challenge of solving a distributed constraint problem is the limited availability of information for each agent (Yokoo et al. 1998). The following are privacy characteristics of a distributed constraint problem (Ismel 2007), (Faltings, Leaute & Petcu 2008).

- Agent Privacy: An agent does not know other agents it does not have a constraint with. From the example in Figure 2.1, Agent A does not know of the existence of Agent C, Agent D or Agent E. Agent B knows of Agent A, Agent C and Agent E but not Agent D. Agent C knows of Agent B, Agent D and Agent E but not Agent A. Agent D knows of Agent C but not Agent A, Agent B or Agent E. Agent E knows of Agent B and Agent C but not Agent A or Agent D.

- Variable Privacy: A variable is only known to the agent that represents it. An agent also knows its variables neighbours, i.e other variables they share constraints with. From the example in Figure 2.1, Agent A knows its variable a together with variable b in Agent B which has a constraint with variable a. Agent B knows its variable b together with variable a in Agent A, variable c in Agent C and variable e in Agent E. Agent C knows its variable c together with variable b in Agent B, variable d in Agent D and variable e in Agent E. Agent D will only knows its variable d together with variable c in Agent C.

- Constraint Privacy: An agent knows only the constraints of the variables it represents. In Figure 2.1, Agent A knows the constraint (a = b). Agent B knows 3 constraints (a = b), (b $\neq$ c), (b = e). Agent C knows 4 constraints (b $\neq$ c), (b = e), (c $\neq$ d), (c $>$ e). Agent D knows the constraint (c $\neq$ d). Agent E knows 2 constraints (b = e), (c $>$ e).

- Domain Privacy: A variable's domain values are known only to the agent that represents it. An agent who owns variables with inter-agent constraints knows only the current assignment of other variables involved in these constraints. For example, in Figure 2.1, Agent A knows variables a's domain D_a = $\{1, 2\}$ and the current assignment of variable b = 1 but not the rest of variable b's domain values or the domain of variables c, d, e belonging to Agents C, Agent D and Agent E respectively.

2.5 Terminology

1. **Assignment:** An assignment is a pair $< x = val >$, where x is a variable of some agent and val is a value from $x's$ domain that is assigned to it.

2. **Cost (utility):** This is used to determine the value (reward) of selecting an assignment.

3. **Consistent assignment:** This refers to the assignment of a value to a variable that satisfies all the constraints the variable is involved in. An assignment is inconsistent otherwise.

4. **Empty assignment:** This is a state where none of the variables in a problem have been assigned a value.

5. **Partial assignment:** This is a state where not all the variables in a problem have been assigned a value.

6. **Complete assignment:** A complete assignment is found when all the variables in a problem have been assigned a value from their domain. It may not necessarily be a solution, i.e. the assignment may not satisfy all constraints.

7. **Connected variables:** Two variables are connected if they share a constraint.

8. **Variable's neighbour:** A variable's neighbour is a variable that is connected (shares a constraint) with it. If the neighbour belongs to the same agent as the variable, it is an internal neighbour while if it belongs to another agent, it is an external neighbour.

9. **Agent's neighbour:** Agent A's neighbour is an Agent B who owns a variable which is related to one of Agent A's variables via an inter-agent constraint.

10. **AgentView:** An AgentView is an agent's knowledge of the current assignments of other agents' variables related to its own variables. Note that an AgentView is not necessarily up to date.

11. **Improvement:** An improvement to the current assignment of a variable is an assignment of another value from its domain that lowers the number of constraints it violates.

12. **Tie:** This refers to a situation where there are two or more options which appear best.

2.6 Chapter Summary

In this chapter, we discussed the Distributed Constraint Satisfaction Problem (DisCSP) framework. DisC-SPs extends Constraint Satisfaction Problems (CSPs) for addressing problem distribution. DisCSPs are hard problems and the goal of a search algorithm is to return a solution that satisfies all constraints (i.e. the aggregated constraint cost is zero). We also discussed DisCSP with CLPs and Distributed Constraint Optimisation Problems (DCOPs) which are natural extensions of DisCSPs. While in DisCSPS, constraints are valued as $\{0, 1\}$ (where 1 refers to a constraint violation and 0 otherwise), DCOPs may have a different value for each constraint and a solution minimises the aggregated constraint costs (a solution may contain some constraint violation). The focus of this thesis is on algorithms for DisCSPs. We also discussed privacy in distributed problems and some terminologies used in this thesis. In the next chapter, we discuss the benchmark problems we use to evaluate the algorithms implemented in this thesis.

Chapter 3

Benchmark Problems

3.1 Introduction

In this chapter, we describe four benchmark problems used to empirically evaluate our algorithms which are: distributed graph colouring problems, random distributed constraint satisfaction problems, distributed meeting scheduling problems and distributed sensor networks problems.

Distributed constraint satisfaction problems that represent several variables per agent have a natural clustering of variables which leads to the problem being seen as a collection of inter-related subproblems or clusters. Each of these clusters forms a Complex Local Problem (CLP) within an agent which has some links to other variables belonging to other clusters (i.e. other agents CLPs). For each problem class, we describe the parameters required. However, when generating naturally DisCSP with CLPs, we modify the DisCSP by making the proportion of internal variables higher than the proportion of external variables and ensure there is a higher number of constraints between variables in the same agent with fewer constraints between variables belonging to different agents. Once the DisCSP has been generated, privacy restrictions are imposed as follows: an agent has knowledge of its own variables, domains and constraints plus the variables (and current values) its CLP is related to via inter-agent constraints. Thus, knowledge about internal variables, their domains and intra-agent constraints is private and is only known to the agent who owns the CLP. In addition, the domains of external variables are also private, although related agents may know some of the values contained in these domains. Privacy in DisCSP is described in Chapter 2, Section 2.4.

This chapter is organized as follows: The distributed graph colouring problem is discussed in Section 3.2. In Section 3.3, we describe random distributed constraint satisfaction problems and distributed meeting scheduling problems are explained in Section 3.4. This is followed by the description of distributed sensor network problems in Section 3.5. The metrics for empirical evaluation is discussed in Section 3.6. Finally, we summarise the chapter in Section 3.7.

3.2 Distributed Graph Colouring Problems

Graph colouring problems have a huge number of applications such as timetabling, sudoku, map colouring and mobile frequency allocation. We generate graph colouring problems as described in (Fitzpatrick & Meertens 2001). In these problems, the concept is to colour the nodes of a graph with a given number of colours ensuring that no two adjacent connected nodes are coloured using the same colour. The colours are the possible values a variable can take. The constraints are *not equal* constraints between two connected variables (nodes) i.e no two adjacent variables should be assigned the same colour. These types of problems are represented by the tuple <*A, n, k, deg*> where:

- A represents the given number of *agents*;
- n represents the number of nodes *(variables)* distributed among the agents;
- k represents the number of colours *(domain size)*; and
- deg represents the connectivity *(constraints)* of the graph.

The degree deg determines the number of edges i.e. the number of constraints in the graph. The constraints are randomly selected and assigned to variables. For example, the parameter setting <5, 50, 3, 5> generates a 3-colour graph colouring problem with 50 nodes distributed among 5 agent and a degree of 5 which means to randomly select and assign 5 constraints to each node i.e. 125 constraints in total. For DisCSPs with CLPs, we use the partitioning method described in (Hirayama, Yokoo & Sycaraw 2004) ensuring that the generated graphs have a higher proportion of intra-agent to inter-agent constraints. Additionally, we ensure a higher proportion of internal to external variables to create naturally distributed problems. Both the intra-agent constraints and inter-agent constraints are *not equal* constraints. Finally, privacy restrictions are then considered ensuring that each agent has complete control of its CLP.

3.3 Random Distributed Constraint Satisfaction Problems

Random problems are used to generate problems with varying characteristics. Random problems (Palmer 1985) are represented by the tuple <*A, n, d,* p_1, p_2 > where:

- A represents the given number of *agents*;
- n represents the number of *variables* distributed among the agents;
- d represents the number of values per variable (*domain size*); and
- p_1 and p_2 are numbers between 0 and 1 which represents the constraint density and constraint tightness respectively.

The constraint density p_1 is used to determine the number of constraints and the constraint tightness p_2 defines the proportion of excluded value pairs by each constraint. The constraints between variables and the excluded value pairs are both chosen at random. For example, a random problem with <*5, 75, 8, 0.4, 0.3*> implies, a problem with 5 agents, 75 variables each with a domain size of 8 which is typically a value from $\{0, ..., 7\}$. The 75 variables are distributed between the 5 agents. The number of variables (n) determines the maximum possible number of constraints as shown in Equation 3.1.

$$maximumConstraints = n * (n-1)/2 \tag{3.1}$$

The *constraint density* of 0.4 translates to 40% of the maximum possible number of constraints used to generate the problem (see Equation 3.2),

$$noOfConstraints = maximumConstraints * 0.4 \tag{3.2}$$

A *constraint tightness* of 0.3 implies 30% of the total possible combination of values are randomly excluded and forbidden (for each pair of variables involved in a constraint).

For DisCSPs with CLPs, we ensure a higher proportion of internal variables compared to external variables. We also ensure that there is an imbalance between the number of intra-agent constraints (within an agent) and inter-agent constraints (those between agents) to create naturally distributed problems.

3.4 Distributed Meeting Scheduling Problems

Distributed meeting scheduling is the problem of determining when meetings between several people possibly from different departments in an organisation will be held (Ismel 2007). Thus, each department has several meetings to schedule. While some meetings involve people within a single department, others involve people in other departments (i.e inter-departmental). Attendees (people) are not part of the problem formulation but the constraints they are involved in are considered.

The *constraints* in a meeting scheduling problem are: (i) precedence constraints to ensure the order in which meetings take place is appropriate; (ii) no two meetings with at least 1 attendee in common are held at the same time (difference constraints); and (iii) Each attendee must have sufficient time to travel to where a meeting will be held (travel time). The travel time between two locations is randomly determined between 0 and the maximum travel time. For two meetings within a department, either a precedence constraint or a difference constraint is randomly selected and added. A travel time is also added between two meetings which are in different departments. Note that meeting timeslots are set to an hour. When considering meetings that should occur at the same time, the difference constraint would be set to zero. Note that overlapping meetings are not allowed as all meetings have a duration of 1.

To generating a meeting scheduling problem instance, we use the tuple *<A, m, d, p, md>* where:

- A represents the set of departments (*agents*) each containing meeting(s) to schedule;
- m represents the set of meetings (*variables*);
- d represents a set of timeslots (*domains*) that indicate when meetings can take place;
- p represents the constraint density; and
- md represents the maximum travel time between meeting locations.

The constraint density is used to determine the number of constraints. For example, to generate a meeting scheduling problem having the parameter setting *<5, 60, 7, 0.18, 3>* this, translates to 60 meetings distributed among 5 departments with 7 timeslots. 0.18 is the constraint density (i.e. 18% of possible number of constraints) and a maximum travel time of 3.

For DisCSPs with CLPs, we use (i) a higher proportion of intra-agent to inter-agent constraints; and (ii) a higher proportion of internal to external variables to create naturally distributed problems. The precedence, difference and travel time constraints also apply to the external variables but only precedence and difference constraints apply to the internal variables.

3.5 Distributed Sensor Network Problems

Sensor DisCSPs are used in applications such as industrial monitoring, detection, tracking and data collection. When generating sensor network problem instances, we use grid-based Sensor DisCSP (Zhang, Wang, Xing & Wittenburg 2005). The objective is to assign a three distinct sensors to the three variables used to track a target while satisfying the visibility and compatibility constraints. The *visibility* of targets to a sensor are the assigned set of sensors that can detect it while the *compatibility* refers to the positioning of these sensors in relation to the targets. Two sensors are compatible if they have a constraint (edge) between them while a set of three sensors are compatible if they are pair-wise compatible and form a triangle. A sensor network problem is formulated as a DisCSP to comprise of *<A, s, pv, pc >* where:

- A represents a set of targets (*agents*)
- Each agent has 3 *variables*, one for each sensor that is required to track the corresponding target;
- The *domain* of each variable is the set of sensors *s* within its visibility.
- pv and pc represents the parameter controlling density of visibility and compatibility *constraints*.

The *intra-agent constraints* are the three sensors assigned to the target must be distinct, they must form a triangle and they must be pair-wise compatible. The *inter-agent constraints* ensures that a given sensor can only be selected by one agent.

For example, to generate a Sensor DisCSPs with <*5, 6, 0.8, 0.6*> implies, a 6 grid sensor network (i.e. 36 sensors) to track 5 targets that have a visibility of 0.8 (80%) i.e 80% of the total number of sensors can possibly detect each target and a compatibility of 0.6 (60%) i.e 60% of the total number of sensors that can detect each target. The fixed (3) number of variables per agent in distributed sensor networks limits the number of possible intra-agent constraints and thus, results in a higher ratio of inter-agent constraints to intra-agent constraints. This inherent feature differentiates them from naturally distributed problems, however, we would use sensor network problems for additional evaluation.

3.6 Performance Metrics

We measure the performance of our algorithms along these three evaluation metrics:

- Number of messages: In solving a DisCSP, each agent exchanges messages with neighbouring agents to check the consistency of its current proposed assignments to variables involved in external constraints. Thus, the total number of messages exchanged between agents is used to measure the communication costs of finding a given solution.
- Number of non-concurrent constraint checks (NCCCs): An approach similar to Lamports logical clocks (Lamport 1978) is used. To determine a measure of concurrent search efforts, each agent keeps a counter of its constraint checks. On receiving a message, an agent then compares and updates its counter with the largest counter among its neighbours (Meisels, Kaplansky, Razgon & Zivan 2002). The NCCCs represents computational time of finding a given solution.
- Percentage of problems solved: The thesis focuses on distributed local search algorithms which by default are incomplete. Thus, we consider the percentage of problems solved as a metric to measure the percentage of problems solved within a specified time (i.e. the maximum number of iterations).

The importance of these metrics depends on the type of problem to be solved and the resources available. With an unreliable system or when the number of agents is high, the number of messages exchanged will be of more concern. However, with fewer agents, the problem becomes more complex and NCCCs would be of concern. In general, a good distributed algorithm will minimise the number of messages and NCCCs as well as solve the highest number of problems.

3.7 Chapter Summary

In this chapter, we described the problems used in evaluating the algorithms in this thesis which are distributed graph colouring problems, random DisCSPs, distributed meeting scheduling problems and distributed sensor network problems. These are benchmark problems that are general abstractions for many real-life problems and are used for evaluating distributed constraint satisfaction algorithms. Distributed graph colouring problems are used to model problems such as timetabling, map colouring and mobile frequency allocation. Random DisCSPs are used to generate random problems while distributed meeting scheduling problem models the distributed scheduling of meetings between the employees of a company. Sensor DisCSPs are used in industrial monitoring and data collection. Unlike distributed graph colouring problems, random DisCSPs and distributed meeting scheduling problems, sensor network problem are not naturally distributed problems because the number of variables per agent is fixed at 3 and this restricts the number of possible intra-agent constraints. However, we use sensor network problems for additional evaluation to determine the performance of our algorithms on such settings. We also listed the metrics used for empirical evaluation. In the next chapter, we discuss search strategies and algorithms for distributed constraint problems.

Chapter 4

Algorithms for solving Distributed Constraint Problems

4.1 Introduction

Research in distributed constraint problems has proposed several problem solving algorithms. Based on the mode of execution of these algorithms, they are categorized as either **synchronous** or **asynchronous** algorithms. Algorithms for solving distributed constraint problems can also be categorized into two groups, namely **local search** and **systematic search**. Hybrid algorithms aim to combine the advantages of the search techniques. In this chapter, we describe these categories and review existing DisCSP and DCOP algorithms.

This chapter is organized as follows. We discuss the search methods in Section 4.2. In Section 4.3, we present a review of distributed constraint satisfaction algorithms. This is followed by a discussion of distributed constraint optimisation algorithms in Section 4.4. Finally, we summarise the chapter in Section 4.6.

4.2 Search Methods

In this section, we differentiate between synchronous and asynchronous algorithms. We also describe systematic and local search algorithms. Some heuristics for improving search in algorithm for solving distributed problems are also discussed.

4.2.1 Synchronous and Asynchronous Algorithms

Algorithms for DisCSPs can also be categorized as either synchronous or asynchronous based on their execution model (Yokoo et al. 1998), (Zivan & Meisels 2003). Synchronous algorithms have a low degree of parallelism between different agents execution and are based on privilege where each agent performs actions in a predefined order. A token is passed among agents with the active agent being the agent in possession of the token. When the active agent completes its processing, the token is then passed to the next agent. The current active agent receives the updated information from the previous agents before proceeding with the search. On the other hand, asynchronous algorithms have a higher degree of parallelism with every agent possibly active at any time.

4.2.2 Systematic Search

Systematic search algorithms (also known as backtracking or complete search) are complete methods, i.e. return the solution for solvable problems and also determine unsolvable problems (Rossi, Beek & Walsh 2006). Systematic search algorithms use an exhaustive search and incrementally attempt to extend a partial assignment (that specifies consistent values for some of the variables) by repeatedly choosing a value for the next variable which is consistent with the values in the current partial assignment. If a consistent assignment does not exists, the search *backtracks* to a previous variable.

More concretely, systematic search algorithms start with an empty partial assignment (known as Current Partial Assignment (CPA)), proceed to select one unassigned variable at a time and assign a consistent value to the variable to form an extended CPA (illustrated in Figure 4.1). The process of selecting an unassigned variable and assigning it a value is referred to as the *forward phase*. A current partial assignment is made up of consistent instantiations of the selected variables. If the next variable is selected but a consistent assignment cannot be found, the search goes to the *backward phase*. In the backward phase, the search backtracks to the previous most recent variable and then proceeds by removing the variable's assignment from the CPA and then trying to extend the CPA with an alternative domain value. If the domain values of the variable are exhausted and no consistent assignment exists for the variable, the search backtracks further. The backtrack stops when a variable with an alternative consistent assignment is found or the first variable is reached with no alternative consistent assignment, thus, the problem has no solution. However,

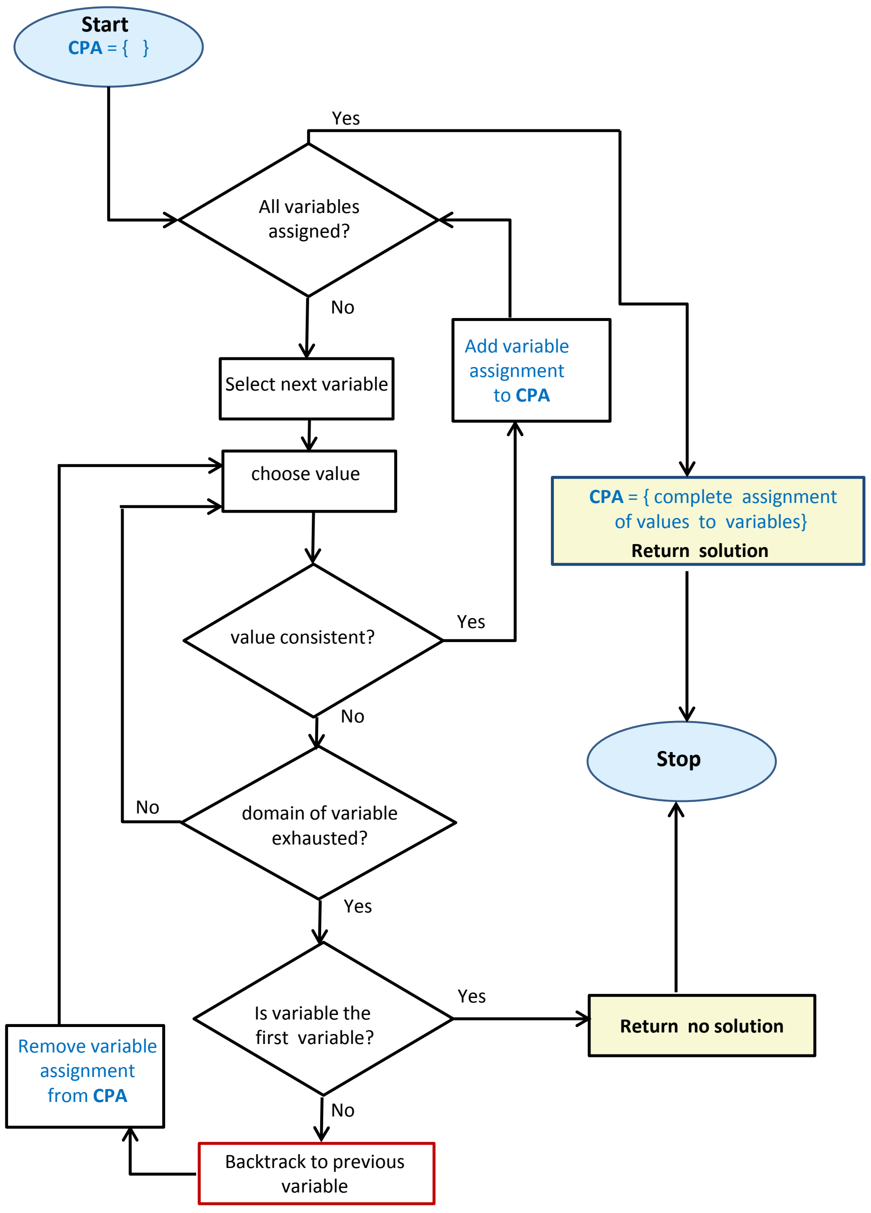

Start
CPA = { }
Yes
All variables assigned?
No
Select next variable
Add variable assignment to CPA
choose value
CPA = { complete assignment of values to variables}
Return solution
value consistent?
Yes
No
Stop
No
domain of variable exhausted?
Yes
Is variable the first variable?
Yes
Return no solution
Remove variable assignment from CPA
No
Backtrack to previous variable

Figure 4.1: Systematic search

if a consistent assignment is found during the backtrack phase, the variable assignment is populated to the CPA and the forward phase commences.

In summary, when a variable assignment is consistent, it is added to the partial assignment otherwise the search backtracks. Backtracking may be done several times. The main weakness of systematic search algorithms is the exponential amount of time they may require which makes them expensive for solving large DisCSPs.

4.2.3 Local Search

Local search (also known as iterative improvement search) is a type of search that iteratively improves an initial, normally inconsistent complete assignment by changing the value of one or more variable(s) at a time. The initial assignments are usually randomly or heuristically generated and in successive iterations, the values assigned to variables are changed so that the number of constraint violations is reduced until a termination condition is met. Local search algorithms are incomplete algorithms thus, they may not find a solution for problems where a solution exists. In a local search algorithm, the "best" available value (leading to the least constraint violations) for a variable is selected, thus, this makes them liable to getting trapped in **local optima** (see below).

Local Optima (Deadlock): Local-optima also known as (deadlock) is a state where variables in some agents are violating some constraints and neither the agent nor any other agent in the DisCSP can make local changes that can result in a lower number of constraint violations. A weaker form of local optima in local search is referred to as **Quasi-Local Optima** (QLO). A QLO is a local optima detected by local communications between an agent and its neighbours (Zhang, Wang & Wittenburg 2002). In a QLO, an agent or its neighbours cannot make local changes that can result in a lower number of constraint violations. However, another agent (not the neighbour of the agent that encountered the QLO) may change its assignment to a value which improves the overall state (reduces the overall number of constraint violations). Thus, when a QLO is detected, it does not necessarily result in local optima but local optima imply that a QLO has also been encountered. Local search algorithms generally differ on the heuristic/strategy used to escape local (quasi-local) optima.

For example, refer to the simplistic problem (in Chapter2, Figure 2.1) of allocating timeslots for 5 student presentations again in Figure 4.2. Each agent (A, B, C, D, E) represents a single variable X = {a, b, c, d, e} respectively. D$_{x_i}$ where $x_i \in$ {a, b, c, d, e} represents the *domain* (timeslots) that can be assigned to each student, which are D$_a$ = $\{1, 2\}$, D$_b$ = $\{1, 2\}$, D$_c$ = $\{1, 3\}$, D$_d$ = $\{1, 3, 4\}$, D$_e$ = $\{2, 3\}$. Constraints between the variables are illustrated with a line between variables. The number of constraint violations for a student's variable value are depicted by Viol$_{x_i}$.

To illustrate a quasi-local optima, if the current assignment of the variables is <a = 1, b = 1, c = 1, d =

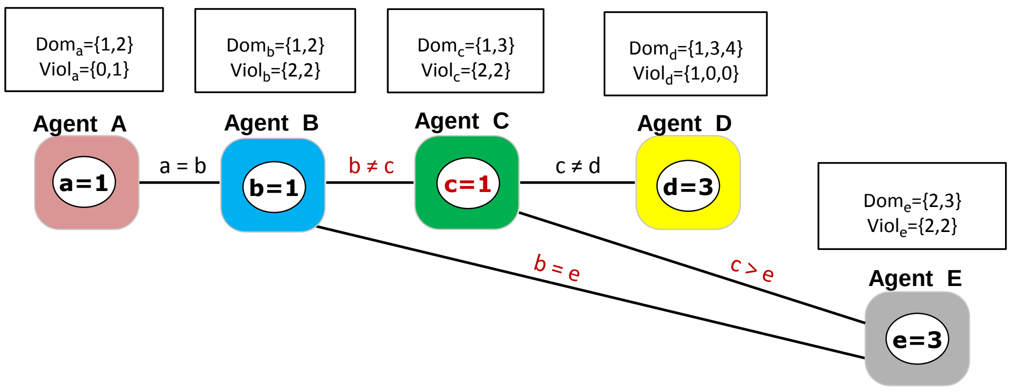


Figure 4.2: A simple illustration of local optima

3, e = 3 $>$ as illustrated in Figure 4.2, Agent C cannot find a value which improves variable c. Variable c's neighbours are variables $\{$b, d, e$\}$. Variable b in Agent B has the same number of constraint violations of 2 for both its domain values, variable d in Agent D is already consistent and variable e in Agent E also has the same number of constraint violations for both the domain values 2 and 3. Thus, Agent C detects a quasi-local optima because it cannot improve the current assignment of variable c and variable c's neighbours have not changed values.

4.2.4 Heuristics for Improving Search

To improve the efficiency of search algorithms, many heuristics have been proposed which can either be used alone or in combination. These include heuristics for escaping local optima and heuristics for ordering values, variables and agents.

- **Strategies for Escaping Local (or Quasi-Local) Optima:** Distributed local search solving techniques use several strategies to escape from local optima. In this section, we discuss strategies for escaping local optima and describe the algorithms that implement some of these strategies in Section 4.3.1.

1. Tabu Search

   Tabu search (Glover 1990) is a heuristic used to learn bad combinations of values. It is used to ensure previously visited non-solutions are not revisited by learning and storing the set of assignments found in a deadlock in a tabu store. At local optima, the tabu store is updated with the most recent bad combinations in order to avoid revisiting those values. The larger the size of the store, the better the heuristic becomes but consequently, the size of the store and the time to check the store can become exponential.

2. Random Restart

   The simplest local search (Dechter 2003) starts with a random instantiation of variables with values and iteratively improves this initial instantiation using local repairs until a deadlock is reached. When a deadlock is encountered, the search is restarted by random re-initialization of value assignments to variables to enable the search to proceed. This process is repeated until a solution is found or the specified number of iterations is reached. A drawback of random restart is that the effort made by the search before the random restart is wasted (not reused) and therefore, the restart is made with no knowledge learnt from the previous search.

3. Weights on Constraints

   The weights on constraints approach for escaping quasi-local optima (Hirayama & Yokoo 2005, Duong, Pham, Sattar & Newton 2013) attaches a weight to each constraint (initially 1). It increases weights attached to constraints violated at local optima to make the satisfaction of those constraints more important. The summation of the weighted number of constraint violations is used to determine the *cost function*.

4. Penalties on Values

   The Penalty driven heuristic (Basharu, Arana & Ahriz 2005) attaches a penalty to each variable's domain value (initially 0). If quasi-local optima are encountered, current domain values leading to constraint violations are penalised, hence, marking those values as bad choices and encouraging the search to find another value within that variable's domain. Penalties are reset under certain conditions. The summation of the violated constraints together with value penalties is used to determine the *cost function*.

5. Probability

At local optima, the search uses some probability to (i) maintain its current assignment or; (ii) select another value with the same number of constraint violations; or (iii) select a value with more constraint violations in order to enable other parts of the solution space to be explored. The determination of the optimal probability can be expensive, it is also problem dependent and can generally determine the performance of the algorithm (Zhang et al. 2002, Zhang et al. 2005).

Simulated annealing - SA (Kirkpatrick, Gelatt & Vecchi 1983) is a probabilistic approach used to find approximate solutions to optimisation problems. SA initially allows some bad moves that are gradually decreased. Thus, the algorithm randomly explores the solution space starting with a large, general search space and gradually, the algorithm concentrates on promising parts of the search space by selecting values "closer" to the solution (i.e. values with less number of constraint violations). This unique form of probabilistic selection enables the avoidance of local optima.

6. Coalition

   Coalition was proposed by (Hirayama & Toyoda 1995) for breaking deadlocks. A coalition is a contract between agents (with one agent voted as manager) that are involved in a local optima with the purpose of satisfying individual or collective constraints. The manager gathers all variables, domains and constraints in a coalition and solves the local problem using either selfish or altruistic strategies. In the selfish approach, values that satisfy violated constraints are selected without any regard to the effect on other previously satisfied constraint that may become unsatisfied while all constraints are considered in the altruistic strategy. A form of coalition is used where agents take turns being mediator (i.e. they centralize small, relevant portions of the problem) in (Mailler & Lesser 2006). Coalition improves search at the expense of privacy.

- **Ordering Heuristics:** Ordering heuristics determine which agents and variables have precedence in the search and which values an agent chooses for its variables first. Thus, it encourages the algorithm to focus the search in more promising or difficult parts of the problem.

  Ordering can be static or dynamic. A static ordering is determined at the start, before the search process commences while dynamic (flexible) ordering occurs during search and the most recent search conditions can be used to control and determine how each agent makes subsequent decisions.

1. Value Ordering

   Some criteria used to order domain values include; (a) Ordering the values by prioritising on the value that removes the smallest number of values from the domains of variables not yet considered; (b) the use of the most recently assigned value in order to capitalize on previous consistent partial assignments; and (c) *Min-Conflict* (Steven, Mark, Andrew & Philip 1992) selects the value with the least number of constraint violations.

2. Variable Ordering

   In Brelaz's heuristic (Brélaz 1979), variables are ordered by the number of constraints with already ordered variables. Ties are broken initially by the number of constraints with currently unordered variables and then lexicographically. Other variable ordering strategies include; (a) The *domain size* of variables as proposed (Haralick & Elliott 1979) in which preference is given to variables with smaller domains; and (b) *maximum degree* is another heuristic (Freuder 1982) that selects the variable involved with the largest number of constraints with other variables not yet considered.

3. Agent Ordering

   The task of ordering agents considers the structure of neighbouring agents together with constraints and domains. Deciding an agent ordering is similar to assigning a priority to the agent. Dynamically changing the priority order of an agent to a higher priority enables the agent to revisit bad decisions early in the search. When the search cannot find a value for a variable that is consistent with the variable values of its higher priority neighbours, the priority of the agent representing the variable is changed so that it has the highest priority among its neighbours (Yokoo & Hirayama 2000). Other heuristics for agent ordering include a measure of the constraint violations for each agent, such that the agent with the highest constraint violations has the highest priority (Zhou, Thornton & Sattar 2003).

4.3 DisCSP Algorithms

4.3.1 Local Search DisCSP Algorithms for Single Variable/Agent

The Stochastic Distributed Penalty Driven Search Algorithm (**Stoch-DisPeL**) (Basharu, Arana & Ahriz 2006) is a synchronous iterative improvement algorithm for solving DisCSPs that escapes local optima with the penalty on value approach. At quasi-local optima, an agent stochastically implements either a temporary penalty or an incremental penalty. A temporary penalty is a fixed value penalty of 3 assigned to perturb the solution to encourage agents to exploit other values in their domain. Temporary penalties are reset immediately an agent selects another value. The incremental penalty is a fixed value of 1 imposed on the assignment of inconsistent variables at quasi-local optima. Incremental penalties are reset when an agent (i) finds consistent assignments for its variables; or (ii) when the search space is distorted by penalties during the search (Basharu et al. 2006). An agent sends messages to its lower priority neighbours (lower priority neighbours that violates a constraint with it), informing them to also impose an incremental penalty (temporary penalty) on their assignments. The cost function for Stoch-DisPeL is determined by the summation of the number of constraint violations together with the penalties imposed.

Distributed Breakout Algorithm (**DBA**) (Yokoo & Hirayama 1996) is a hill climbing algorithm which extends the breakout algorithm (BA) (Morris 1993) to distributed problem solving. In DBA (later renamed Single Distributed Breakout (**SingleDB**) (Hirayama & Yokoo 2005)), a weight is associated with each constraint and is initially set to 0. The cost function is determined by the number of the weighted constraint violations. In this thesis, we refer to the algorithm as SingleDB. An agent sends its initial assignments to its neighbours via *OK messages*. Agents then carry out a distributed steepest search by exchanging possible improvements to a candidate solution. An agent sends its initial assignment to its neighbours via *OK messages* and waits for all of the *OK messages* issued by its neighbours before selecting an assignment for its variable that would reduce the number of constraint violations (also called an *"improve"*). This is then sent in a *Send Improve message* to its neighbours. On receiving these messages, an agent then compares its improve with the improve of neighbours and the agent with the best proposed improvement is given the right to change its value. To break a tie, i.e. when more than one related agents have the same best improvement, the agent with the highest ID is selected. When an agent encounters quasi-local optima, the weights attached to constraints violated are increased by 1. Two stochastic variations; SingleDB weak probability (DBA-wp) and SingleDB strong probability (DBA-sp) that differ in the approach at tie breaking were later proposed (Zhang & Wittenburg 2002). In DBA-wp, ties are broken such that both agents may implement the change or not, or just one agent implements the change while in DBA-sp, an agent implement the change if it has the best improvement among its neighbours but when it can improve (not the best improvement among its neighbours), it will implement the change based on a probability. Later, SingleDB was improved with a weight decay mechanism in SingleDB-wd (Lee 2010). At each iteration in SingleDB-wd, before computing possible improvements, agents update their constraint weights as follows: (i) Weights on violated constraints at time t are computed as $W_{i,t} = (dr * W_{i,t-1}) + lr$; and (ii) Weights on satisfied constraints at time t are decayed as $W_{i,t} = \{max(dr * W_{i,t-1}), 1\}$ where: d_r is the decay rate ($d_r < 1$) and l_r is the learning rate ($l_r > 0$).

The Distributed Stochastic Algorithm (**DSA**) (Zhang et al. 2002) is a synchronous hill climbing algorithm that implements a probabilistic strategy where agents use a probability to change their value when making that change will improve their solution. The choice of the probability in DSA can be problem specific and difficult to determine. The summation of the number of constraint violations is used to determine the cost function in DSA. Also, message sending between agents is minimised because messages are sent only when an agent changes its variable's value. Several variations of DSA exist that differ in the strategies for value change (Zhang et al. 2002) used to escape quasi-local optima. Some particularly dominant strategies are seen in DSA-A and DSA-B. In DSA-A, with probability p, an agent may change to the value that gives the most reduction in the number of violated constraints or keep the current value unchanged with probability ($1 - p$). DSA-B is similar to DSA-A but with the same probability (p), an agent may also take another equally good value without making the solution better or worse. Distributed Probabilistic Protocol (**DPP**) (Smith & Mailler 2010) is a hybrid of the DSA and SingleDB protocols that dynamically implement randomness and direct control which is based on the structure and current state of the problem.

4.3.2 Local Search DisCSP Algorithms for Multiple Variables/Agent

Distributed Penalty Search for Agents with Multiple Local Variables (**Multi-DisPeL**) (Basharu, Arana & Ahriz 2007b) is an algorithm (that extends Stoch-DisPeL) for solving problems with complex local problems that implements the penalty on value approach. Multi-DisPeL uses a steepest descent search for the local problem and Stoch-DisPeL is used for combining each agents solutions into a complete solution for the problems (i.e. satisfy inter-agent constraints).

SingleDB is also extended to handle complex local problems in Distributed Breakout Algorithm for

Agents with Multiple Local Variables (**Multi-DB**) (Hirayama & Yokoo 2002), (Hirayama & Yokoo 2005). Initially, Multi-DB was modified to the Distributed BreakOut (**DisBO**) (Eisenberg 2003), increasing constraint weights only at local optima where no agent can change to result in an improved state. Later, (Basharu et al. 2007b) proposed Distributed BreakOut weight decay (**DisBO-wd**) which uses a weight decay mechanism. At each step in DisBO-wd, an agent decreases the weights on all the constraints its variables are involved in and additionally, increases the weights of violated constraints (as described in SingleDB-wd above).

4.3.3 Systematic Search DisCSP Algorithms for Single Variable/Agent

A well-known complete method proposed for solving DisCSPs is synchronous backtracking **(SBT)** (Yokoo et al. 1998) which uses a fixed ordering of variables and agents take turns to select a value. A Current Partial Assignment (CPA) containing variables with their current consistent assignment is sent to the next agent. A backtrack is done when an agent cannot find a consistent value for a variable from its domain. Later, an asynchronous backtracking algorithm (**ABT**) (Yokoo & Hirayama 2000) that allows agents to search in parallel for a solution was proposed. Agents have a fixed *priority* which is determined by the order of the agent's identifier (ID) and an agent with a higher ID implies the agent has a higher priority. Agents asynchronously assign a consistent value to their variable, sending *OK messages* with their value to neighbour agents to update their *Agentview* and then evaluate their assignments (by computing the number of constraints violated). For a given agent with an inconsistent value and no consistent value in its domain, the agent records the conflict in its *no-good store* and backtracks to the lowest priority agent with a higher priority than itself (informing it that its assignment does not extend to a solution). If the conflict involves other unrelated (via contraints) agent(s), the agent creates a new link (a not equal constraint between the variables and their assignments in the CPA) and then sends a new link request (*nogood message*) to the unrelated agent(s) asking them to also create the new constraint. The addition of new constraints in ABT was eliminated to increase privacy and this elimination sometimes results in a trade-off on performance (Bessière, Brito, Maestre & Meseguer 2005). Distributed Intelligent Backtracking (DIBT) (Hamadi 1998, Hamadi 2002) is a distributed asynchronous graph based algorithm that also avoids no-good recording. DIBT proposes an intelligent backtrack technique using parallel exploration of search trees and constraint propagation. Knowledge of the conflict of previous searches is also saved to enable a conflict directed backtracking.

The authors of ABT presented Asynchronous Weak-Commitment Search (**AWCS**) (Yokoo 1995a). AWCS is a complete asynchronous backtracking algorithm that dynamically prioritises agents favouring backtracking agents. In AWCS, each agent concurrently assigns a value to its variable, and sends the value to other neighbouring agents. After that, agents wait for and respond to incoming messages. Similar to ABT, agents in AWCS are assigned a priority but unlike ABT where priority is fixed, an agents priority is changed dynamically. Initially, priorities are set to 0 and a higher priority value implies the agent has a higher priority. An agent in AWCS determines values from its domain that satisfy all constraints its variable has with their higher priority neighbours, and from these values it selects a value for its variable that minimises constraint violations with their lower priority neighbours. When an agent does not find a consistent assignment, the agent sends no-good messages to notify its neighbours and then increases its priority by 1. The priority increases the importance of satisfaction of the variable represented by an agent.

A complete algorithm that uses partial centralization (a form of coalition) is Asynchronous Partial Overlay (**APO**) (Mailler & Lesser 2003), (Mailler & Lesser 2006). Agents in APO take turns being mediators to solve the relevant parts of the problem using cooperative mediation. In mediation, agents centralize some part of their problem and use explicit constraint and domain sharing whenever requested. This approach results in some privacy loss.

Distributed Knowledge-Based Hybrid Approach (**DisHyb**) is a two phase approach that combines distributed local and distributed systematic search (Lee, Arana, Ahriz & Hui 2009a). In the first phase of the algorithm, a distributed local search algorithm is run to gather knowledge about difficult variables and values. The agents are then reordered according to the knowledge about difficult variables and values. In the second phase, a distributed systematic search algorithm is run with the agents reordered. Penalty-Based Distributed Hybrid Algorithm (**PenDHyb**) and Weight-Based Distributed Hybrid Algorithm (**DBHyb**) are two implementations of the approach that differ on the local search used. Stoch-DisPeL is used in PenDHyb while SingleDB-wd is used in DBHyb.

4.3.4 Systematic Search DisCSP Algorithms for Multiple Variables/Agent

Asynchronous Weak-Commitment Search for Agents with Multiple Local Variables (**Multi-AWCS**) (Yokoo 1995a) and Asynchronous Backtracking for Agents with Multiple Local Variables (**Multi-ABT**) (Hirayama et al. 2004) are complete algorithms that extend their single variable per agent version. Both Multi-AWCS and Multi-ABT use a local AWCS (ABT) solver to satisfy the intra-agent constraints and a global AWCS

(ABT) solver to satisfy the inter-agent constraints.

A version of Multi-ABT with an alternative no good learning and value selection techniques was also produced (Maestre & Bessière 2004).

Multi-Hyb (Lee et al. 2009a) and **Multi-HDCS** (Lee, Arana, Ahriz & Hui 2009b) use a hybrid of complete and local search for solving DisCSPs with CLPs. Multi-Hyb extends the idea in DisHyb for DisCSPs with CLPs. A centralised systematic search was used to solve the CLPs while a distributed local search and a distributed systematic search concentrate on the global problem. Similar to DisHyb, a two phase approach is used. In the first phase, a distributed local search algorithm is run to gather knowledge about difficult variables and values in the global problem. Then in the second phase, a distributed systematic search is run to solve the global problem using agents that have been reordered according to the variables and values learnt by distributed local search. However, Multi-HDCS differs from Multi-Hyb by running the distributed local search and distributed systematic search concurrently. Thus, the approach uses a single phase and the distributed systematic search runs at the same time as the distributed local search and centralised systematic searches. Two implementations of each framework are presented that differ on the local search algorithm used (Lee 2010). Penalty-Based Hybrid Algorithm for Agents with Multiple Local Variables (**Multi-Hyb-Pen**) and Weight-Based Hybrid Algorithm for Agents with Multiple Local Variables (**Multi-Hyb-DB**) are two implementations of Multi-Hyb that use Stoch-DisPeL and SindleDB-wd respectively. While Penalty-Based Hybrid Distributed Concurrent Search Framework for Agents with Multiple Local Variables (**Multi-HDCS-Pen**) and Weight-Based Hybrid Distributed Concurrent Search Framework for Agents with Multiple Local Variables (**Multi-HDCS-DB**) are two implementations of Multi-HDCS that use Stoch-DisPeL and SindleDB-wd respectively.

4.4 DCOP Algorithms

4.4.1 Local Search DCOP Algorithms for Single Variable/Agent

In coordination algorithms (Maheswaran, Pearce & Tambe 2004), a small group of agents coordinate their action based on their local constraints. The quality of the solution is measured by the cost of coordination between agents against the quality of the solution reached. Hence, theoretical guarantees on the quality of solution (that defines the *optimality* of the solution) can be provided (Pearce & Tambe 2007), (Vinyals, Shieh, Cerquides, Rodriguez-Aguilar, Yin, Tambe, & Bowring 2011). In surveying the literature on local search DCOP algorithms, we differentiate algorithms with and without guarantee on the quality of solution.

Algorithms Without Guarantee: A very different approach towards local search is implemented in **Max-Sum** (Farinelli, Rogers, Petcu & Jennings 2008). Max-Sum is a synchronous, incomplete DCOP algorithm that uses message-passing and belongs to the class of algorithms referred to as generalized distributive law (Aji & McEliece 2000). Max-Sum operates on a factor graph, i.e. a bipartite graph where the node represents a variable function and a constraint function. A *variable-node* is connected to all function-nodes that represent constraints with which it is involved in. A *function-node* is connected to all variable-nodes involved in the constraints it represents. The variable-nodes accumulate the costs received from function neighbours and send messages to the function-nodes about the cost received from all its neighbours, excluding that of the receiving function. The function-node on the other hand selects the best combination assignment and also selects a proposed value for that variable. At the end of an iteration, each variable-node selects the value assignment with the best cost based on the messages received from all neighbouring function-nodes.

Anytime Local Search for DCOP (**ALS**) (Zivan 2008) is a framework proposed to enhance any DCOP local search algorithm with the anytime property (i.e. at any given time, the best solution is returned). In ALS, a Breadth-First Search (BFS) is used with one agent designated as the *root agent*. The root agent is used to determine the global solution by calculating the best state for each agent and if found, propagate the new best step. More specifically, an agent (i) receives messages from its neighbours which are lower in the ordering about the quality (cost) of the solution found; (ii) the agent then includes its own state; (iii) calculates the cost with respect to the objective function; and (iv) sends messages to its parents and root agent.

Algorithms With Guarantee: SingleDB (Single Distributed Breakout) and DSA (Distributed Stochastic Algorithm) for DisCSPs were initially extended to 1-Optimal algorithms for DCOPs by (Zhang et al. 2002) in **MGM-1** and **DSA-1** respectively. MGM-1 (Maximum Gain Message), an asynchronous algorithm unlike SingleDB, removes the breakout technique for escaping local optima and the search focuses only on improve (gain) message passing. Similar to SingleDB, on receiving the assignments of all its neighbours, an agent in MGM-1 computes its local improvement (i.e. a reduction in cost) and sends this proposal to its neighbours. After collecting the proposed reductions from its neighbours, an agent changes its assignment only if it has the best proposed reduction compared to its neighbours. Later, DSA-1 and

MGM-1 were further extended to 2-Optimal algorithms **MGM-2** and **SCA-2** (Stochastic Coordination Algorithm)(Maheswaran et al. 2004) in which optimization was done by agents acting in pairs in order to maximise their combined cost.

The authors further generalised these algorithms to k-Optimal algorithms (MGM-k and SCA-k) in which a group of k agents can coordinate value updates in order to maximize their combined cost (Maheswaran et al. 2004). K-Optimal algorithms avoid local optima to which a smaller group would have converged. Thus, the solution found cannot be improved by k or fewer agents changing their values, k being the number of agents.

Another form of optimality, *t-Distance* defines optimality based on a group of surrounding variables within a fixed distance t of a central variable. T-Distance algorithms return *t-optimal* solutions that cannot be improved by the variables within t. Distributed Asynchronous Local Optimization (DALO) (Kiekintveld, Yin, Kumar & Tambe 2010) is an algorithm that can compute either k-Size or t-Distance solutions for any settings of k or t. Each group is assigned a leader that computes a new optimal assignment for the group and implements the new assignment if it is an improvement.

Later, *c-Region* (Vinyals et al. 2011) optimality was introduced to provide quality guarantees beyond size and distance. This provides an alternative trade-off to size and distance, with agents being more aware of the complexity of the regions they generate. Similarly, *c-optimal* solutions cannot be improved by any other assignment inside region c. DALO was extended for region optimality by modifying how agents create their groups in C-DALO (Vinyals et al. 2011).

Theoretical guarantees to the quality of solutions for Max-Sum (see above) were later provided in **bounded Max-Sum** (Rogers, Farinelli, Stranders & Jennings 2011). In Bounded Max-Sum, the problem is made less complex by removing constraints which have the least impact on the solution quality, producing a new problem. Max-Sum is simultaneously used to solve the new problem and to compute the approximation ratio i.e. how close the solution found is to the actual solution. Later, an improved bounded Max-Sum was proposed which uses a tighter approximation ratio (Rollon & Larrosa 2012).

Divide-and-Coordinate Sub-gradient Algorithm (**DaCSA**) (Vinyals, Pujol, Rodriguez-Aguilar & Cerquides 2010), Divide-and-Coordinate by Egalitarian Utilities (**EU-DaC**) (Vinyals, Rodríguez-Aguilar & Cerquides 2010) and Decomposition with Quadratic Encoding to Decentralize (**DeQED**) (Hatano & Hirayama 2013) are algorithms based on the divide and coordinate framework. In the *divide stage*, each agent creates a copy of the problem based on its Agent's view, updates its own sub-problem with information received from its neighbours and solves this updated sub-problem. In the *coordinate stage*, each agent sends some information to its neighbours. The algorithms differ in the encoding method implemented in the divide stage and in the type of messages exchanged between agents.

4.4.2 Local Search DCOP Algorithms for Multiple Variables/Agent

Dynamic Complex Distributed Constraint Optimization Problem - **DCDCOP** (Khanna, Sattar, Hansen & Stantic 2009) is a local search algorithm for solving dynamic DCOPs that handles complex sub-problems by finding local optimal solutions through a branch and bound whilst determining the global optimal solution through the combined calculated best found cost of the agents (including their inter-agent constraint). Multi Variable Multiply Constrained MGM (**MV-MC-MGM**) (Portway & Durfee 2010) is an extension of MGM-1 for MV-MC-DCOPs. MV-MC-MGM implements a decomposition approach for dealing with multiple variables per agent. Each agent calculates the best move for its agents given the values of their neighbours while ensuring that local limiting cost functions are not exceeded.

4.4.3 Systematic Search DCOP Algorithms for Single Variable/Agent

ABT is extended for solving Max-CSPs where a solution minimises the number of constraint violations (Hirayama & Yokoo 2000). Synchronous Branch and Bound (**SynchBB**) (Hirayama & Yokoo 1997) is the distributed version of the branch and bound method (Freuder & Wallace 1992) for solving Max-CSPs. In branch and bound, the search algorithm explores branches of its tree by checking its accumulated cost against an upper and lower estimated bounds on the optimal solution. Partial assignments are represented by a Current Partial Assignment (CPA). For a proposed assignment, if the lower bound is less than the upper bound, the value is accepted and the search commences. Otherwise, another value is checked. If all the values are not accepted, the search backtracks. SynchBB was later extended to address ADCOPs in SynchABB (Grinshpoun et al. 2013).

OptAPO (Mailler & Lesser 2004) is an extension of APO for optimisation problems. In OptAPO, agents share the knowledge of their problems to improve their local solutions.

Asynchronous Forward-Bounding (**AFB**) (Gershman, Meisels & Zivan 2006) is also based on branch and bound, where agents asynchronously propagate CPAs. Taking turns, an agent adds its assignment in the CPA and sends a Forward Bounding-CPA message *(FB-CPA message)* to agents without an assignment in the CPA. On receiving a *FB-CPA message*, an agent calculates the lower bound and returns it to the sender agent via a Forward Bounding Estimate message *(FB-ESTIMATE message)*. For a proposed assignment, if the lower bound is less than the upper bound, the value is accepted and the search commences. Otherwise, another value is checked. If all the values are not accepted, the search backtracks. Asymmetric Two Way Bounding (**ATWB**) (Grinshpoun et al. 2013) is a version of AFB proposed to deal with ADCOPs.

Asynchronous Distributed Constraint Optimization with quality guarantees (**ADOPT**) (Modi et al. 2005) is the first complete DCOP algorithm that allows asynchronous concurrent execution and is guaranteed to terminate with the globally optimal solution. Agents in ADOPT are initially ordered in a Best-First Search (BFS) tree structure called a *pseudo-tree*. Each agent has a *context* (equivalent to an AgentView) consisting of the set of <variable=value> assignments of other preceding agents. A *parent-child relationship* exists among agents that are directly connected via a constraint. An agent higher in the ordering is a parent and child otherwise. When an agent receives a message, it does the following; processes it, sends value messages and sends cost messages. *Value messages* are sent to its children and pseudo-children when an agent changes its value. This enables them to make informed decisions in their selection. *Cost messages* consisting of the context, lower bound and upper bound are sent from children to parent. Additionally, *threshold messages* are sent from parents to children i.e a message containing the lower bound of the cost that the children have from the previous search. ADOPT was extended in BnB-ADOPT (Yeoh, Felner & Koenig 2014). The main difference with ADOPT is that the BFS in ADOPT is substituted with a DFS (Depth First Search) branch and bound approach. Thus, while an agent in ADOPT assumes a value that minimizes its lower bound, an agent in BnB-ADOPT assumes a value when it is able to ascertain that the optimal solution for that agent is probably no better than the cost of the best solution found. Secondly, while an agent in ADOPT uses thresholds to store previously computed lower bound for its current context, thresholds are used to store the cost of the best solution found and also as a means to efficiently make change in BnB-ADOPT. ADOPT-ng (Silaghi & Yokoo 2006) unifies ADOPT with asynchronous backtracking to improve the communication model with a valued no-good protocol. Messages sent in BnB-ADOPT and ADOPT-ng are small in size but can be exponential in number compared to messages sent in ADOPT.

Dynamic Programming Optimization Protocol (**DPOP**) (Petcu & Faltings 2005) is a DFS synchronous, inference DCOP algorithm that exchanges cost functions. A DFS tree is constructed similar to ADOPT. *Utility (cost) messages* are sent bottom up (i.e from child to parent) while *value messages* are sent top down (i.e from parent to child). In the utility phase, an agent combines the costs received from its children and parents to project its cost which is then passed to its parents. While in the value phase, the root finds a value that minimises the costs received and informs the children. Although both utility and messages sent in DPOP are linear in number, utility messages can be exponentially large both in memory and size.

4.4.4 Systematic Search DCOP Algorithms for Multiple Variables/Agent

ADOPT (Modi et al. 2005) was extended to address CLPs using aggregation, interchangeable local assignments and relaxation techniques in (Burke & Brown 2006b), (Burke & Brown 2007). ADOPT for Multiple Variables per Agent (**AdoptMVA**) was also proposed (Davin & Modi 2006) as an extension of ADOPT designed to specifically handle multiple variables per agent. Agents are ordered and centralized search procedures are used to solve each agents' sub problem with distributed ADOPT for the overall problem.

4.5 Discussion

In this thesis, we are concerned with solving large naturally DisCSPs with local search algorithms (for easy problems other existing approaches may be best). Naturally distributed problems expect more internal constraints than external constraints with the problems distributed in clusters. Although local search algorithms are incomplete, they are popular on large DisCSPs, converging quicker to a solution when compared to systematic algorithms (Rossi et al. 2006). A major drawback of local search algorithms is it propensity of getting stuck at local optima i.e. a non-improving state (or deadlock). Significant researches have focused on heuristics which attempt to either escape or avoid local optima. The heuristics are discussed in 4.2.4 and the local search algorithms for DisCSP having single variable per agent that implement the heuristics in 4.3.1.

We identified three of these heuristics; weights on constraints implemented in SingleDB-wd (Hirayama & Yokoo 2005), penalties on values implemented in Stoch-DisPeL (Basharu et al. 2005) and dynamic agent prioritisation implemented in AWCS (Yokoo 1995a). SingleDB-wd, Stoch-DisPeL are distributed local search algorithms while AWCS is a distributed systematic algorithm for single variable per agent. Each of these heuristics focuses on a different part of the DisCSP (<Agent, variables, domain, constraints>). Weights on *constraints* makes the satisfaction of the constraints more important. Penalties on *domain values* is finer grained and allows for more drastic contortion in the cost landscape moving the search to part of solution space where a solution may be found. Dynamic *agent* prioritisation increases the importance of an agent over its neighbours.

Constraint weight escapes local optima and enables the search to proceed but the empirical evaluation reported by the authors [1] of penalty-based approach, showed that the number of agents changing value and becoming more consistent were found to be more with the penalty-based approach when compared to constraint weights (Basharu, Arana & Ahriz 2007a). AWCS outperforms the gold standard in systematic DisCSP algorithms (ABT (Yokoo et al. 1998)). While agent priorities are static in ABT, AWCS introduces dynamic prioritisation of agent which changes the authority of decision making and increase the chances of finding quicker solutions (Yokoo 1995b).

Although these heuristics escape local optima, a challenge in designing local search algorithms remains in strategies for escaping local optima. In order to encourage collaboration and knowledge sharing, a combination of heuristics that identify difficult constraints, bad values and dynamically prioritises the satisfaction of variables an agent represent may be more effective.

4.6 Chapter Summary

In this chapter, we discussed systematic and local search techniques for solving distributed constraint problems. These algorithms can execute in either a synchronous or asynchronous mode. On large problems, local search algorithms are faster compared to systematic search algorithms. However, the major drawback of local search is its propensity of getting trapped at a quasi-local optima. To address this drawback, local search algorithms use heuristics such as constraint weight, value penalties and dynamic agent prioritisation to escape from local optima. A literature review of DisCSP and DCOP algorithms for single variable per agent and multiple variables per agent were also presented.

In the next chapter, we investigate combination heuristics in DisCSPs where an agent represents one variable.

[1] we replicated some of the experiments and found results that confirm the claims

Chapter 5

Combination Heuristics in Local Search for DisCSPs with One Variable/Agent

5.1 Introduction

Local search algorithms for DisCSPs generally differ on the heuristic used to escape from local optima. Escaping local optima enables the search to proceed. The heuristics possibly direct the search to a good solution space thus, leading to quicker solutions. In Chapter 4, Section 4.2.3 we discuss local optima and heuristics for escaping local optima. Against this background, the main focus of this chapter is to investigate the effect of combining existing heuristics for solving DisCSPs with a single variable per agent in an attempt to answer the research question: *"Can a combination of existing heuristics be more effective in escaping local optima?"*.

This chapter is organized as follows. In Section 5.2, we briefly discuss the most effective existing heuristics considered and the proposed combination strategies. Section 5.3 describes Dynamic Agent Prioritisation with Penalties (DynAPP), an algorithm which implements the combination of penalties on value and dynamic agent prioritisation heuristics and was found to perform best overall. Experimental results on instances of several problem classes are presented in Section 5.4. Finally, we summarise in Section 5.5.

5.2 DisCSP Combination Heuristics

To investigate the benefits of combining heuristics in local search for DisCSPs with single variable per agent, we use heuristics which take into account difficult constraints, bad values combinations and dynamic agent prioritisation. Next we discuss constraint weights, penalties on value and dynamic agent prioritisation heuristics implemented in DisCSP and then introduce our combination heuristics.

Constraint weight is used to escape quasi-local optima by increasing the weights on violated constraints by 1. Initially implemented in SingleDB (Hirayama & Yokoo 2005) and later SingleDB-wd (Lee 2010), which is an improved variation of SingleDB where a weight decay mechanism is used to decrease constraint weights at every iteration thus “forgetting” weights over time.

Penalty on value (Basharu et al. 2006). At a quasi-local optima, the current assignment of violating variable are penalised. This aims to identify and avoid bad values that resulted in the quasi-local optima and possibly leads to the selection of another “better” value.

Dynamic agent prioritisation (Yokoo 1995a) increases the importance of an agent over its neighbours (i.e. the satisfaction of constraints that variable is involved in is more important that the satisfaction of constraints involving only lower priority neighbours.) when the agents variables are involved in a quasi-local optima.

The reader is referred to Section 4.3.1 for a discussions on SingleDB-wd and Stoch-DisPeL and Section 4.3.3 for a discussion on AWCS.

Constraint weight and penalty on value are heuristics implemented in distributed local search algorithms while dynamic agent prioritisation has been implemented in a distributed systematic search. Used individually, these heuristics have been found to be effective in escaping/avoiding local optima in DisCSP algorithms. The heuristics tend to modify the search space to allow for a better exploration of other parts of the search space. Our hypothesis is that in combining these heuristics we would further improve how local optima are handled thus improving the efficiency of distributed local search algorithms. The proposed combination strategies are discussed next.

5.2.1 Proposed Combination Strategies

We combine the following heuristics: (i) constraint weights and value penalties; (ii) perform a multi-context search with constraint weights and value penalties; and (iii) combine dynamic agent prioritisation and value penalties. Preliminary results of combining dynamic agent prioritisation with constraint weights gave no benefit and thus are not presented.

1. **Combination of Constraint Weights and Value Penalties:** In these combinations, both constraint weights and value penalties heuristics are implemented in a distributed local search. We add the weighted constraint violations and the value penalties to determine the cost of selecting a given value. Additionally, either of the two heuristics is used to break a tie.

2. **Multi-Context Search:** In the multi-context search, two local search algorithms; SingleDB-wd and Stoch-DisPeL are ran concurrently in a master-slave architecture with restart. Both algorithms solve the problem by randomly initialising from different parts of the search space. The master solver is responsible for solving the problem and terminating the search i.e. returning a solution if found or returning no solution found otherwise. However, the purpose of the slave is to provide the master with its best found assignments. After a number of iterations, the master algorithm checks with the slave algorithm if a "solution" with a lower number of constraint violations exists and re-initialises its variables with the value assignments received from the slave. This approach aims to improve the overall search performed by the master solver.

3. **Combination of Dynamic Agent Prioritisation with Penalties on Values:** This is an approach which combines dynamic agent prioritisation and penalties on variables values. The penalties on values are used to escape quasi-local optima while the dynamic agent priority is used to dynamically change the priority of satisfying the variable represented by the agent and its neighbours, thus, favouring "difficult" variables.

4. **Dynamic Agent Prioritisation with Constraint Weights:** Two heuristics combined are dynamic agent prioritisation and constraint weights as follows: (i) A priority is attached to each agent; and (ii) weights are attached to each constraint. At quasi-local optima, the weights on violated constraints are increased and the priority of the agent is also changed.

5.2.2 Discussion

We use the combination heuristics to implement distributed local search algorithms for DisCSPs where an agent represents a single variable. Algorithms that combine constraint weights and values penalties and multi-context search are presented in Appendices.

- We implement the following algorithms with the constraint weight and value penalties combination heuristics: Stochastic Distributed Penalty Driven Search with Weight (**Sweight**), Single Distributed Breakout with Weight Decay and Penalties (**Dpenalties**), Normalised Stochastic Distributed Penalty Driven Search with Weights (**normSweight**), Stochastic Distributed Penalty Driven Search with Probabilistic Tie Breaking (**StochTBP**), Stochastic Distributed Penalty Driven Search with Tie Breaking (**StochTB**) and Stochastic Distributed Penalty Driven Search with Weights for tie breaking cost (**costwTB**). These algorithms differ on the heuristic used to determine the cost function and the heuristic used to break a tie.

- We implement Single Distributed Breakout with Weight Decay and periodic restart (**SingleDB-wd+Restart**) and Stochastic Distributed Penalty Driven Search with periodic restart (**Stoch-DisPeL+Restart**) with the multi-context search,. In StochDisPeL+Restart, Stoch-DisPeL is the *master solver* and SingleDB-wd is the *slave solver* while in SingleDB-wd+Restart, SingleDB-wd is the *master solver* and Stoch-DisPeL is the *slave solver*.

- We implement **DynCW**, an algorithm that combines dynamic agent prioritisation with constraint weights.

- We implement **DynAPP**, an algorithm that combines dynamic agent prioritisation with penalties on values (DynAPP performed the best and is presented in this chapter).

In an empirical evaluation on several problem classes, we found that the algorithms with the combination heuristics performed better when compared to the existing algorithms that use only one of the heuristic i.e local search algorithms, SingleDB-wd and StochDisPeL. Note: Dpenalties and DynCW solved less than 50% for most problem settings while using an exponential number of messages and NCCCs. To highlight the performance of algorithms that solved more problems, Dpenalties and DynCW were not included in our illustrations of the results as it would obscure the differences in performance for the other algorithms.

At the phase transition, where the problems are more difficult, DynAPP, StochTB, StochTBP and costwTB performed better than Stoch-DisPeL and SingleDB-wd on distributed graph colouring problems for both NCCCs and number of messages. Similarly on random DisCSPs, StochTB, Sweight and DynAPP performed better than both Stoch-DisPeL and SingleDB-wd. Improvements over both Stoch-DisPeL and SingleDB-wd were also seen in StochTB, costwTB and DynAPP on distributed meeting problems. However, overall, we found DynAPP generally performed best.

Similarly, we found that SingleDB-wd+Restart and Stoch-DisPeL+Restart benefited from the restart and performed better than SingleDB-wd and Stoch-DisPeL respectively. When compared to SingleDB-wd+Restart and DynAPP, Stoch-DisPeL+Restart outperformed SingleDB-wd+Restart and performed closely to DynAPP although less prominent.

Overall, DynAPP, the algorithm that combines dynamic agent prioritisation and value penalties performed best and is presented in detail in this chapter. Algorithms that combine constraint weights and values penalties is presented in Appendix A while the multi-context search s presented in Appendix B.

5.3 Dynamic Agent Prioritisation with Penalties

Dynamic Agent Prioritisation with Penalties (**DynAPP**) is a synchronous distributed local search algorithm for solving DisCSPs where an agent represents a single variable with the objective of finding the first solution that satisfies all constraints simultaneously. In DynAPP, two heuristics are combined; dynamic agent prioritisation and value penalty as follows: (i) A priority is attached to each agent; and (ii) penalties are attached to individual domain values. Next, we describe the operations in DynAPP before providing a more detailed description of the algorithm.

5.3.1 DynAPP Operations

Initialization

In DynAPP, each agent has a single variable and is identified by an alphabetic number (ID). Each domain value is associated with a *penalty* and initially all penalties are set to zero. Each agent has a *priority* and initially priorities are determined by the alphabetic ordering of an the agents' ID. The lower the ID, the higher the priority. Each agent randomly assigns a value for its variable, and sends the value and its priority to all its neighbours. On receiving initial messages from all its neighbours, each agent updates their *AgentView* (i.e. their knowledge of the current assignment and priority of all their neighbours).

State Evaluation and Value Selection

An agent determines the cost of each of its domain values and selects the value with the lowest cost. The cost is determined using the minimum conflict (Minton, Johnston, Philips & Laird 1992) heuristic to minimise the function in Equation 5.1. Therefore, a domain value is selected if it minimises the sum of constraint violations and penalties for the chosen value. When two values have the same best cost (i.e. a tie is encountered), the value ordering is used to break the tie, selecting the first value.

$$f(d_j) = viol(d_j) + p(d_j) \qquad {}_j \in \{1, ..., |domain|\} \tag{5.1}$$

where d_j is the ${}_j$th value in the domain

$viol(d_j)$ is the number of constraints violated if d_j is selected

$p(d_j)$ is the penalty currently imposed on d_j with both $viol(d_j)$ and $p(d_j)$ having equal weight.

Deadlock Detection and Escape Strategy

A *quasi-local optima* is encountered when an agent's *AgentView* does not change in two consecutive iterations and there are constraint violations. When a *quasi-local optima* is detected the following steps are taken;

1. *Penalties on Values:* Firstly, a diversification of the search is encouraged by penalising values which lead to constraint violations. Two types of penalties termed; *temporary* and *incremental* penalties are used (similar to Stoch-DisPeL (Basharu et al. 2006)). Penalties make the value undesirable and "hopefully" cause the selection of another value. Penalties are initially set to zero and are increased during the search. The temporary penalty is a fixed value 3 while the incremental penalty is a value 1 [1]). At a deadlock, an agent randomly selects either a temporary or incremental penalty and imposes it on its current assignment. When a **temporary penalties** is imposed, an agent attempts to find combinations of values that can resolve the deadlock. Temporary penalties are reset to zero as soon as they are used and sent to the appropriate neighbour(s). **Incremental penalties** have a lesser chance of finding another value compared to temporary penalties but are allowed to build up to enable agents to learn to avoid selecting values repeatedly associated with deadlock. Incremental penalties are reset to zero during the search when all constraints are satisfied or when penalties distort an agents' cost functions. *Penalty distortion* is detected when the current assignment appears best in a variable's domain but there is another value in the domain, which has fewer number of constraint violations than the current assignment. Thus, this can drive the search away from promising regions of the

[1]These values were found to perform best based on an empirical evaluation (done in (Basharu et al. 2005).

search space. Resetting of penalties by the originator agent does not affect the penalty requests sent to neighbours. Prior to selecting a value, the receiving agent imposes these penalties, searches for another value and resets the penalties based on the reset conditions described above. An agent sends penalty messages to "relevant" neighbours (which differ depending on the situation) requesting the imposition of penalties on their current values. Temporary penalties are sent to lower priority neighbours violating constraints with the agent while incremental penalties are sent to all lower priority neighbours (see 5.3.2 for more on message sending).

2. *Dynamic Agent Prioritisation:* Secondly, an agent's priority is also dynamically changed (similar to AWCS (Yokoo 1995a)). At a *quasi-local optima*, an agent searches for a neighbour with the highest priority with whom it has a constraint violation. If such a neighbour exists, it changes its priority to the priority of the neighbour. Additionally, the agent sends a priority decrease request to its neighbour whose priority it now assumes to reduce its priority by 1 (discussed in more details in Section 5.3.2). However, if the agent has the highest priority among its neighbours, the agent maintains its current priority. Priorities do not affect the ordering of agents but they determine the neighbours of an agent that receive and impose a penalty request.

Communication

Apart from the messages sent at the beginning about the initial random assignment of an agent's variable (see *Initialization*), agents in DynAPP send messages to their neighbours in order to communicate their current assignments, priorities, penalty request and priority decrease request. Although the exact content of a message is discussed in Section 5.3.2, **Send Message**, a typical message may contain the following;

1. The current value of the variable an agent represents.
2. The current agent priority.
3. A penalty request.
4. A priority decrease request.

Algorithm 1 DynAPP: procedure **main**()

```
1: let $c_i$ be the current value of the variable in $A_i$, $i \in \{1..n\}$, randomly selected initially
2: let $p(c_i)$ be penalty on $c_i$
3: let $A_i$.neighs be $A_i$'s neighbours
4: let $msg_{ngb}$ be a message from $A_i$'s neighbour, $A_{ngb}$
5: p = 0.3, TEMPPEN = 3, INCPEN = 1
6: for each agent $A_i$, $i \in \{1..n\}$ concurrently do
7:     $p(c_i)$ = 0, $A_i.penaltyRequest$ = null
8:     $A_i.neigh_{hpvc}$ = null, $A_i.priorityDecRequest$ = FALSE
9:     $A_i$.sendMessages($c_i$,$A_i.priority$,$A_i.penaltyRequest$,$A_i.priorityDecRequest$,
       $A_i.neigh_{hpvc}$)
10: end for
11: while (termination condition not reached) do
12:     for each agent $A_i$, $i \in \{1..n\}$ concurrently do
13:         while ($\exists$ neigh $\in A_i$.neigh | $\nexists$ msg m $\in A_i.inbox$ , m sent by neigh) do
14:             wait
15:         end while
16:         for each $msg_{ngb} \in A_i.inbox$ do
17:             $A_i$.updateAgentView($c_{ngb}$,$A_{ngb}.priority$,$A_{ngb}.penaltyRequest$,
                $A_{ngb}.priorityDecRequest$)
18:             $A_i.inbox \backslash \{msg_{ngb}\}$
19:         end for
20:         $A_i$.chooseValue($c_i$,$A_i.priority$,$A_i.penaltyRequest$,$A_i.priorityDecRequest$,
            $A_i.neigh_{hpvc}$,TEMPPEN,INCPEN,p)
21:         $A_i$.sendMessages($c_i$,$A_i.priority$,$A_i.penaltyRequest$,$A_i.priorityDecRequest$,
            $A_i.neigh_{hpvc}$)
22:     end for
23: end while
```

5.3.2 Algorithm Details

DynAPP is depicted in Algorithms 1 - Algorithm 4. The search begins with the initialisation phase where each agent randomly assigns a value to its variable from its domain and the priority of each agent is determined by the alphabetic ordering of their ID. Each agent then sends the initial value and initial priority to its neighbours (see Algorithm 1, line 6-10). In successive iterations, each agent performs the following;

- **Collects Messages and Updates Agent:** An agent collects messages from its neighbours. When more than one penalty request is received by an agent, only one penalty request is imposed giving priority to an incremental penalty over a temporary penalty. Similarly, when an agent receives more than one priority decrease request, only one request is imposed i.e. the first priority decrease request received. The values and priorities received (see Algorithm 1, line 16-19) are used to update an agent's *AgentView*.

- **Chooses Value:** Next, an agent searches for a value for its variable (see Algorithm 1, line 20) in **chooseValue** (as listed in Algorithm 2) with the lowest cost. Prior to this, an agent carries out the following:

    1. Algorithm 2, line 3-5 - Resets incremental penalties if an agent is either consistent or the cost function is distorted by penalties.
    2. Algorithm 2, line 6-8 - If there is a penalty request, impose the penalty - **doRequest** (listed in Algorithm 3).

        (a) Implement the appropriate penalty if a penalty request has been received (Algorithm 3, line 1-5).

        (b) If a priorityDecRequest has been received, an agent decreases its priority and resets priorityDecRequest to False (Algorithm 3, line 6-9).

        An agent then evaluates the cost of selecting each of its domain values and selects a value with the lowest cost (using the minimum conflict heuristic) in **doMinConflict** (see Algorithm 2, line 11). The cost function to be minimised by the search for each variable is described in Equation 5.1 (see State Evaluation and Value Selection above).
    3. If an agent did not receive a penalty request and an agent's previous and current *AgentView* are not the same (see Algorithm 2, line 10), an agent evaluates the cost of selecting each of its domain values and selects another value with the lowest cost - **doMinConflict** (see Algorithm 2, line 11).
    4. If a quasi-local optima is encountered, it is tackled (described in Section 5.3.1, Deadlock Escape Strategy and listed in Algorithm 2, line 13-26) as follows:

        (a) Firstly, an agent searches for and changes its priority to the priority of the neighbour with the highest priority with which it shares a constraint violation (see Algorithm 2, line 13-16). The neighbour agent with the highest priority is identified (see Algorithm 2, line 15). When two agents have the same highest priority, the first neighbour agent with the highest priority is selected. Also, if no neighbour with a higher priority (i.e. an agent has the highest priority amongst its neighbours) is found, an agent does not change its priority.

Algorithm 2 DynAPP: procedure **chooseValue**(c,priority,penaltyRequest, priorityDecRequest,neigh$_{hpvc}$,TEMPPEN,INCPEN,p)

```
1: let neighs$_{vc}$ be neighbours violating constraints with self
2: let hp be neighbour with highest priority in neighs$_{vc}$
3: if (cost function is distorted or consistent) then
4:     resetPenalties()
5: end if
6: if ($penaltyRequest$ ! = null) then
7:     doRequest($c, priority, penaltyRequest, priorityDecRequest$,TEMPPEN,INCPEN)
8:     doMinConflict()
9: else
10:     if (agentView(t) ! = agentView(t-1)) then
11:         doMinConflict()
12:     else        (quasi-local optima)
13:         if (hp ! = null) then
14:             $priority$[self] = $priority$[hp]
15:             neigh$_{hpvc}$ = hp
16:         end if
17:         select random value r $\in \{0..100\}$
18:         if (r < p*100) then
19:             $penaltyRequest$ = “IncPen”
20:             $p(c) = p(c)+$ INCPEN
21:         else
22:             $penaltyRequest$ = “TempPen”
23:             $p(c) = p(c)+$ TEMPPEN
24:         end if
25:         doMinConflict()
26:     end if
27: end if
```

(b) Secondly, an agent imposes a temporary penalty (with probability p) or an incremental penalty otherwise (see Algorithm 2, line 17-24).

(c) Finally, an agent selects a new value for its variable - **doMinConflict** (see Algorithm 2, line 25).

- **Sends Message:** Messages are then sent (see Algorithm 1, line 21) to the appropriate neighbours as listed in Algorithm 4 and described next.

1. If an agent had encountered a quasi-local optima, a penalty was imposed and an agent has changed its priority, the message sent to the neighbour with the highest priority that was assumed by the agent, the priorityDecRequest is set to True and the message will consists of

Algorithm 3 DynAPP: procedure **doRequest**(c,priority,penaltyRequest, priorityDecRequest, TEMPPEN,INCPEN)

```
if ($penaltyRequest$ = "IncPen" ) then
    p(c)= p(c) + INCPEN
else
    p(c)= p(c) + TEMPPEN
end if
if ($priorityDecRequest$) then
    $priority = priority$[self]+1
    $priorityDecRequest$ = FALSE
end if
```

$< value, priority, null, TRUE >$ (see Algorithm 4, line 7). The message sent to the other neighbours depends on the type of penalty.

(a) If the penalty imposed by an agent is an incremental penalty, it informs its lower priority neighbours to also impose an incremental penalty and the message will consists of $< value, priority, PenaltyRequest, FALSE >$ (see Algorithm 4, line 11).

(b) Otherwise if the penalty is a temporary penalty, the agent informs its lower priority neighbours that it violates a constraints with, to also impose a temporary penalty and the message will consists of $< value, priority, PenaltyRequest, FALSE >$ (see Algorithm 4, line 16).

2. If a deadlock was not encountered, an agent sends a message consisting of only its new value and priority $< value, priority, null, FALSE >$ to all its neighbours (see Algorithm 4, line 21).

These messages are used by the receiving agents to update their *AgentView* and impose the appropriate request. This process continues until consistent values are found (with no constraint violations) or the maximum number of iterations is reached.

5.3.3 Example of DynAPP Execution

Recall the DisCSP example in Chapter 2, Figure 2.1 that represents the simplistic problem of allocating presentation timeslots to 5 students presented again in Figure 5.1. The students are the *variables* X = {a, b, c, d, e}. Agent A, Agent B, Agent C, Agent D and Agent E each represents a single variable a, b, c, d, e respectively. D_{x_i}, $x_i \in$ {a, b, c, d, e} represents the *domain* (timeslots) for each student, which are D_a = {1, 2}, D_b = {1, 2}, D_c = {1, 3}, D_d = {1, 3, 4}, D_e = {2, 3}. Constraints between the variables are illustrated

Algorithm 4 DynAPP: procedure **sendMessages**(c,priority,penaltyRequest, priorityDecRequest,neigh$_{hpvc}$)

```
let neighs be an agents neighbours
let neighs$_{lp}$ be neighbours with lower priority than self
let neighs$_{lpvc}$ be neighbours in neighs$_{lp}$ violating constraints with self
for each $neigh \in neighs$ do
    if ($penaltyRequest$ != null) then
        if ($neigh \in$ neigh$_{hpvc}$) then
            send message($c, priority, null, TRUE$) to neigh
        else
            if ($penaltyRequest$ = “IncPen”) then
                if ( $neigh \in$ neighs$_{lp}$ ) then
                    send message($c, priority, penaltyRequest, FALSE$) to neigh
                end if
            else
                $penaltyRequest$ = “TempPen”
                if ( $neigh \in$ neighs$_{lpvc}$ ) then
                    send message($c, priority, penaltyRequest, FALSE$) to neigh
                end if
            end if
        end if
    else
        send message($c, priority, null, FALSE$) to neigh
    end if
end for
neigh$_{hpvc}$ = null
resetTempPenalties()
```

with a line between variables. Constraints in red are violated. The number of constraint violations for each domain value is depicted by Viol$_{x_i}$ while *Pen*$_{x_i}$ represents penalties imposed on each domain value respectively. The *constraints* are (i) (a = b), i.e. a and b must be assigned the same slot; (ii) (b $\neq$ c), i.e. b and c cannot be assigned the same slot; (iii) (b = e), i.e. b and e must be assigned the same slot; (iv) (c $\neq$ d), i.e. c cannot be assigned the same slot as d; and (v) (c $>$ e), i.e. c must be assigned a time slot after e.

Using this problem to explain DynAPP’s execution, the priority of each agent is determined by the alphabetic ordering of their ID such that Agent A has priority 1, Agent B has priority 2, Agent C has priority 3, Agent D has priority 4, Agent E has priority 5. Thus, the highest priority 1 is assigned to the first agent A. Agent A does not have a neighbour with a higher priority and has Agent B as lower priority neighbour; Agent B has Agent {A} as higher priority neighbour and Agents {C, E} as lower priority neighbours; Agent C has Agent {B} as higher priority neighbour and Agents {D, E} as lower priority neighbours. Agent D

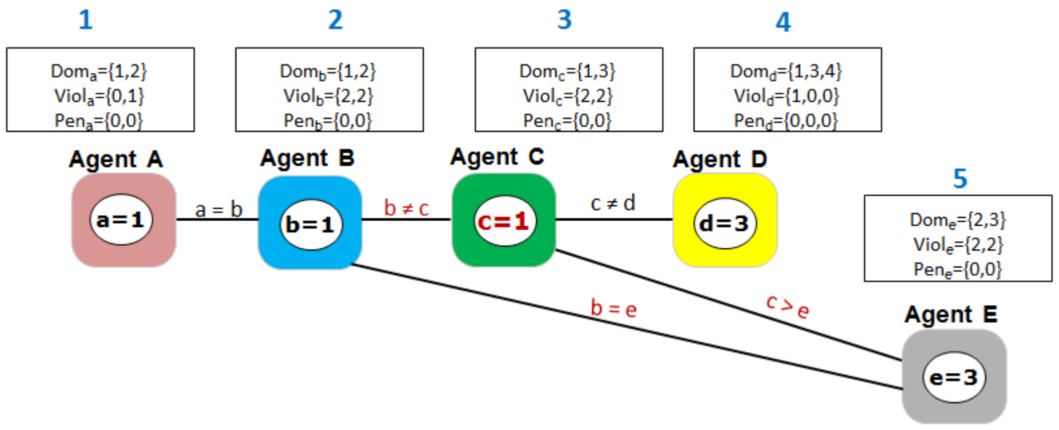


Figure 5.1: DynAPP: example of quasi-local optima

has Agent {C} as higher priority neighbour and no lower priority neighbour. Agent E has Agent {B, C} as higher priority neighbour and no lower priority neighbour.

Agents take turns to respond to messages and send messages to their neighbours. At the assignment of variables <*a = 1, b = 1, c = 1, d = 3, e = 3*>, assume that Agent A and Agent B have taken turns in selecting a value for their variables and it is Agent C's turn to evaluates its state. Variable c in Agent C has the same number of constraint violations for both values in D_c = {1, 3}. Its neighbour Agents {D, E}, variable d in Agent D is consistent and variable e in Agent E also has the same number of constraint violations for both values in D_e = {2, 3}. Agent C detects quasi-local optima. Firstly, Agent C checks its *AgentView* for a conflict neighbour with a higher priority and finds Agent B. Thus, Agent C changes its priority to 2 (i.e. that of Agent B which is the conflict neighbour with the highest priority). Secondly, Agent C imposes a temporary penalty on variable c's value {1}. The new lower priority neighbours of Agent C are Agents {B, D, E}. Consequently, Agent C then assigns the value 3 (which currently has the least cost) to variable c.

Agent C then sends messages to Agent B, Agent D and Agent E. For example, the message sent to Agent B consists of < *value = 3, priority = 2, penalty request = TempPen, priorityDecRequest = True*> because it violates constraint (b≠ c) and to reduce its priority by 1. The message sent to Agent D consists of < *value = 3, priority = 2, penalty request = null, priorityDecRequest = False*> because c's current value is consistent with d's current value.

The message sent to Agent E consists of < *value = 3, priority = 2, penalty request = TempPen, priorityDecRequest = False*> because c's current value violates the constraint c > e with e's current value.

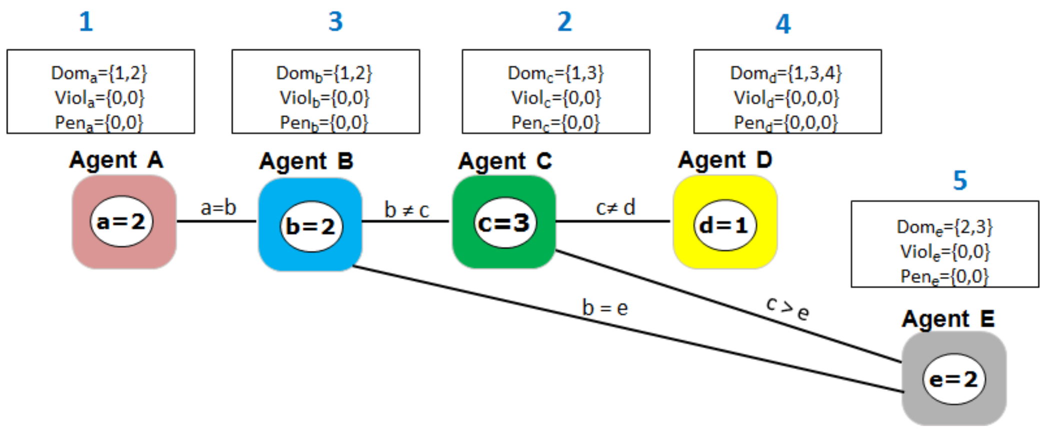


Figure 5.2: DynAPP: example of problem solved

Subsequently Agent D may select 1 or 4 for variable d, Agent E selects 2 for variable e, Agent A selects 2 for variable a and Agent B selects 2 for variable b and the problem is solved (see Figure 5.2).

5.3.4 DynAPP Algorithmic Properties

Termination: The termination detection mechanism in DynAPP is incorporated into the search algorithm. DynAPP (like Stoch-DisPeL) terminates when (i) all agents are consistent and retain their current values in two successive iterations; or (ii) when the maximum number of iterations is reached.

Completeness: DynAPP is an incomplete local search algorithm and cannot determine if a problem is unsolvable. Also, it is not guaranteed to find a solution to a problem even if a solution exists.

Privacy: In DynAPP, each agent only knows its variable (and its domain and constraints) together with its neighbour variables, the current assignment of the neighbour variables and the constraints relating them. It does not know other constraints (or other domain values) its neighbour variables are involved in or other variables (or their constraints and domains) represented by other agents. An agent only reveals the current assignment of its variable to their neighbours.

5.4 Empirical Evaluation of DynAPP

We empirically evaluate DynAPP on **distributed graph colouring problems**, **random DisCSPs** and **distributed meeting scheduling problems** (refer to Chapter 3 for the description of the problems). Because DynAPP is an algorithm for single variable per agent, we exempt sensor network problems from our evaluation due to its inherent nature of having 3 variables per agent. For each problem setting, we run 100 solvable [2] problem instances from which we record and present (i) the percentage of problems solved within the maximum number of iterations; (ii) the median number of Non-Concurrent Constraint Checks (NCCCs) performed; and (iii) the median number of messages sent. Although CPU time is not an established measure for DisCSPs (Meisels et al. 2002) and is not reported in this thesis, we measured CPU time and found the results to be consistent with the other evaluation metrics used.

DynAPP was compared with two other local search algorithms that have a strategy for escaping quasi-local optima - **Stoch-DisPeL** and **SingleDB-wd** and a systematic search - **PenDHyb**. We excluded DSA (Zhang et al. 2002) due to fact that it has no explicit mechanism for escaping from quasi-local optima, thus, does not perform well where the goal is to satisfy all constraints. Once stuck at quasi-local optima, the sideways moves are usually insufficient to push a search out of locally optimal regions. Each algorithm was allowed to run for a maximum of (100 * (n)) iterations where n represents the number of variables (nodes, variables or meetings). Note: when an algorithm did not solve a problem, the messages "wasted" in that problem are not counted towards the median number of messages, they do however affect the percentage of problems solved. We further used run length distribution (**?**) to study how the probability of finding a solution changes with the number of iteration compared to other local search algorithms. PenDHyb is a complete search algorithm and is not included for evaluation on iteration bound problems.

5.4.1 Results on Distributed Graph Colouring Problems

We use 3-colour distributed graph colouring problems with different characteristics. We present results for range of number of nodes are n $\in$ (180, ..., 300) in steps of 20 with a degree of 4.9 (which is at the phase transition representing the region of difficult problems). For each node size, we run 100 problem instances and the results are presented in Figure 5.3.

[2] We use only solvable problem for our evaluation because the algorithms we implement are distributed local search algorithms and they do not detect unsolvability

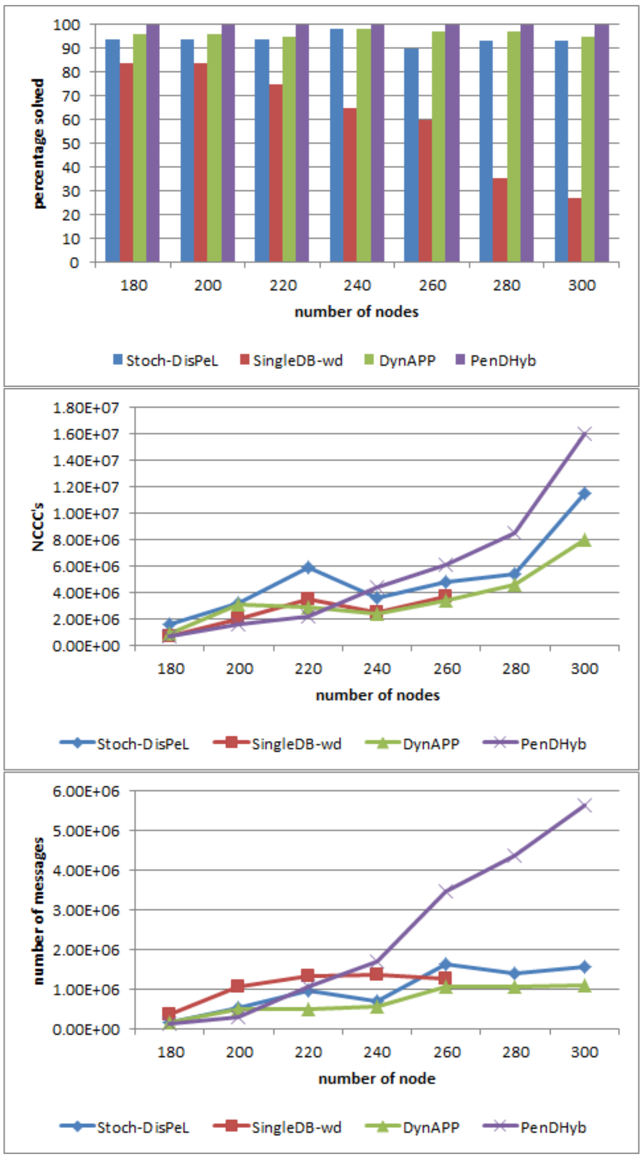


Figure 5.3: $< \text{n} \in \{180, ..., 300\}, |\text{colours}| = 3, \text{deg} = 4.9 >$

DynAPP substantially outperforms Stoch-DisPeL and SingleDB-wd by solving the most problems. PenDHyb is a systematic search algorithm and as expected it solved all the problems. For 180 and 200 variables, Stoch-DisPeL and DynAPP performed closely but as the number of nodes increases, DynAPP performed best, having the least number of messages and NCCCs overall.

We run the local search algorithm on graph colouring problem with the following parameters; $< n = 250, |colours| = 3, deg = 4.9 >$. We run 100 instances of the problem and present the results in Figure 5.4.

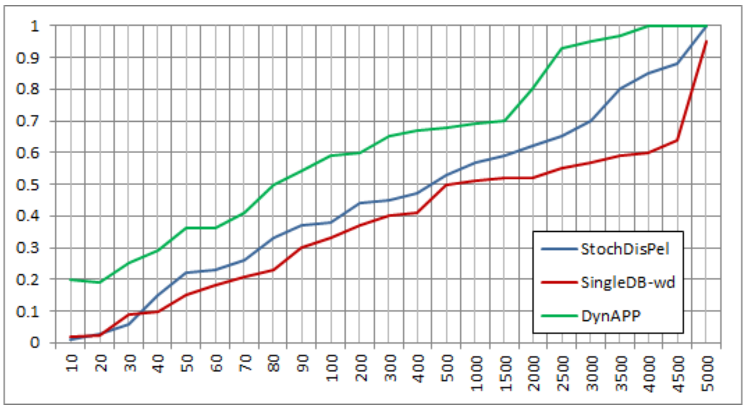


Figure 5.4: $< n = 250, |colours| = 3, deg = 4.9 >$

At incremental iterations, DynAPP had the highest probability of finding a solution compared to Stoch-DisPeL and SingleDB-wd.

5.4.2 Results on Random Distributed Constraint Satisfaction Problems

We use random DisCSPs at the phase transition i.e. the region of hard problems using a constraint tightness p_2 of 0.4, a constraint density p_1 of 0.15 and a domain size of 8. We present results where we vary the number of variables $n \in (90, ..., 200)$ in steps of 10. For each problem size, we run 100 problem instances. The results are presented in Figure 5.5 with DynAPP solving the most problems compared to Stoch-DisPeL and SingleDB-wd.

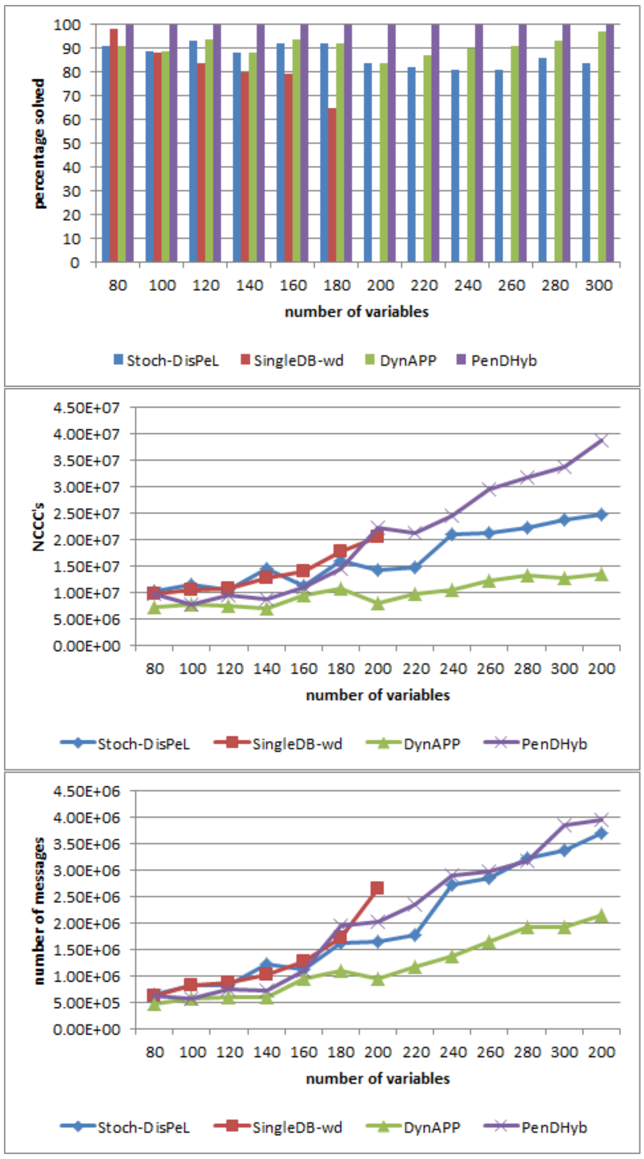


Figure 5.5: $< n \in \{90, ..., 200\}, |\text{domain}| = 8, p_1 = 0.15, p_2 = 0.4 >$

PenDHyb is a systematic search algorithm and as expected it solved all the problems. SingleDB-wd solved less than 40% of the problems from n = 140. To highlight the performance of algorithms that solved more problems, the SingleDB-wd results for 140 or more variable were not included in our illustrations of the results as it would obscure the differences in performance for the other algorithms. We observe that, as the number of variables increases, the number of messages exchanged and the NCCCs for DynAPP, Stoch-DisPeL and SingleDB-wd increased. DynAPP had the least for both messages and NCCCs.

We run the local search algorithm on graph colouring problem with $< n = 200, |\text{domain}| = 8, p_1 = 0.15, p_2 = 0.4 >$. We run 100 instances of the problem and present the results in Figure 5.6.

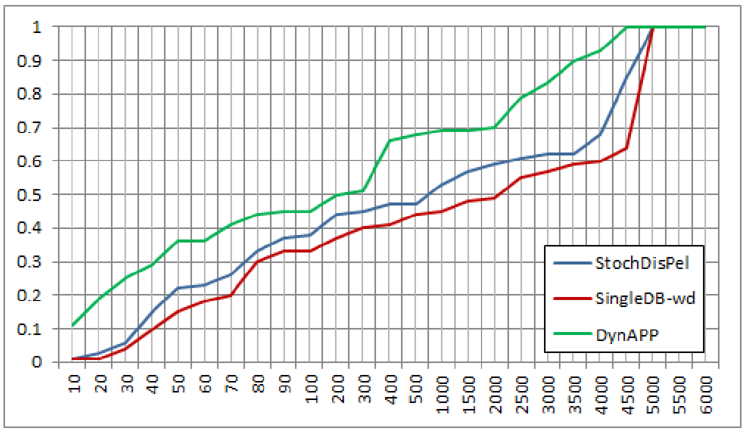


Figure 5.6: $< n = 200, |\text{domain}| = 8, p_1 = 0.15, p_2 = 0.4 >$

At incremental iterations, DynAPP had the highest probability of finding a solution compared to Stoch-DisPeL and SingleDB-wd.

5.4.3 Results on Distributed Meeting Scheduling Problems

To evaluate DynAPP on meeting scheduling problems, we generated problems with varying number of meetings $n \in \{60, ..., 200\}$ in steps of 20. We use 7 timeslots, a maximum travel time (md) of 3 and a constraint density of 0.2. For each meeting size, we run 100 problem instances and the results are presented in Figure 5.7.

On the number of problems solved, DynAPP and Stoch-DisPeL solved the same number of problems (except for 140 meetings, Stoch-DisPeL solved one problem more than DynAPP) with SingleDB-wd solving the least problems.

PenDHyb is a systematic search algorithm and as expected it solved all the problems. For the number of messages and NCCCs, DynAPP generally had the least compared to Stoch-DisPeL, PenDHyb and SingleDB-wd. SingleDB-wd had very large number of messages. To highlight the performance of algorithms that solved more problems and used less number of messages, SingleDB-wd was not included in our illustrations for the number of messages as it would obscure the differences in performance for the other algorithms.

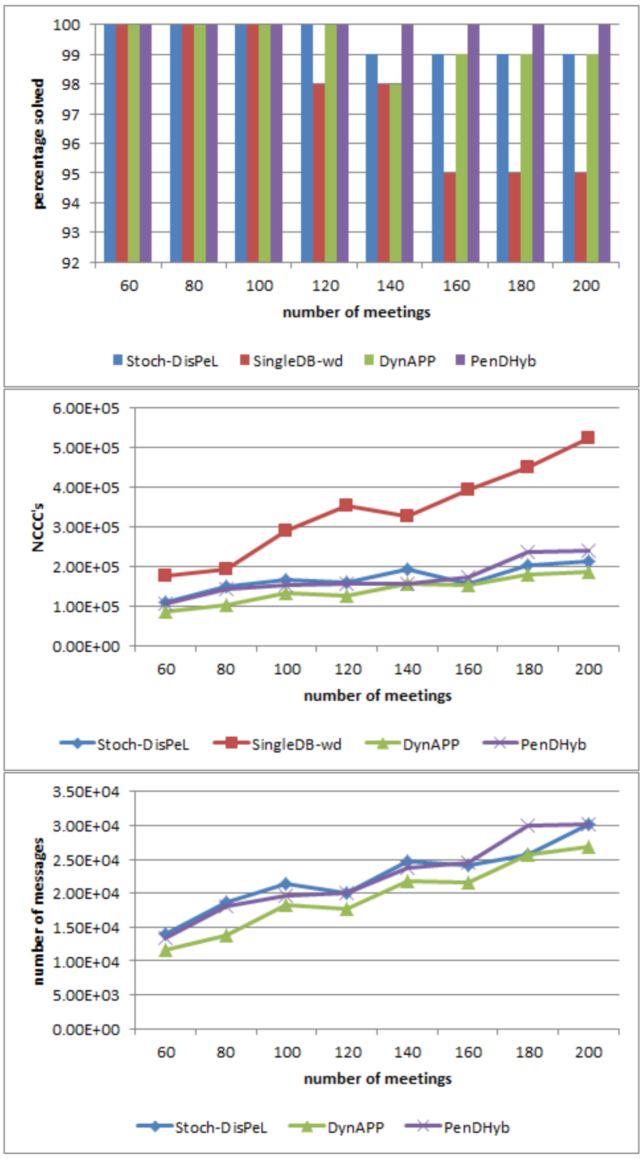


Figure 5.7: $< n \in \{60, ..., 200\}$, timeslots = 7, d = 0.2, md = 3 $>$

We run the local search algorithm on graph colouring problem with $< n = 200, |domain| = 8, p_1 = 0.15, p_2 = 0.4 >$. We run 100 instances of the problem and present the results in Figure 5.8.

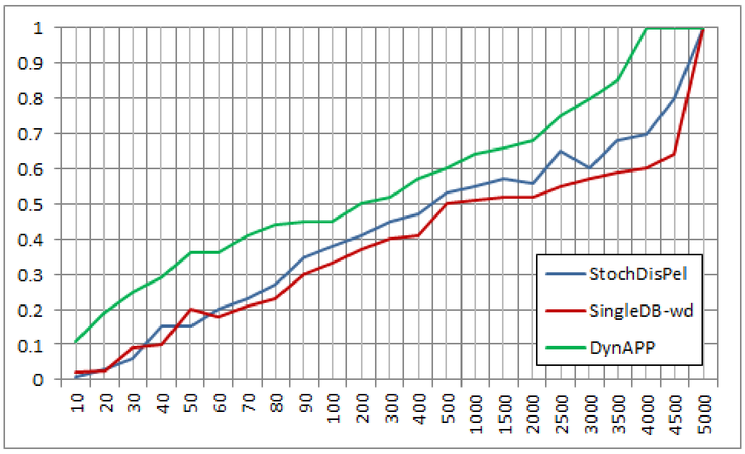


Figure 5.8: $< n = 200, |domain| = 8, p_1 = 0.15, p_2 = 0.4 >$

Similar to distributed graph colouring and random DisCSPs, DynAPP had the highest probability of finding a solution at incremental iterations compared to Stoch-DisPeL and SingleDB-wd. Stoch-DisPeL and SingleDB-wd overlap initially when iteration was less that 60.

5.5 Chapter Summary

In this chapter, we investigated the combination of three heuristics; constraint weights, penalty values and dynamic agent prioritisation. We were interested to see whether a combination of the heuristics may be more beneficial in escaping local optima. In this regard, we carried out the following: (i) we combined constraint weights and value penalties; (ii) we performed a multi-context search with constraint weights and value penalties; and (iii) combined dynamic agent prioritisation with penalty on value in local search algorithms for DisCSPs where an agent represents a single variable. In an empirical evaluation on distributed graph colouring problems, random DisCSPs and distributed meeting scheduling problems, we found that the algorithms with the combination heuristics performed better when compared to existing algorithm that use only one of the heuristic. We focused on DynAPP (an algorithm that combines penalties on values and agent priority heuristics) in this chapter as it gave the best results and present the other combination heuristics at the appendices.

DynAPP solved the most problems while having the least number of messages and NCCCs on a range of DisCSPs. The following contributions have been made:

1. We have combined several existing heuristics in distributed local search algorithms for single variable/agent which were found to improve search when compared with algorithms that use only one of the combination heuristics.

2. DynAPP - a distributed local search algorithm that combines dynamic agent prioritisation with penalties in local search algorithms for DisCSPs where an agent represents a single variable.

"Indeed the combination strategies are more effective for escaping quasi-local optima in distributed local search for single variable per agent settings". In the next chapter, we explore problem-structure oriented problem solving for DisCSPs with complex local problems i.e. where an agent represents multiple variables.

Chapter 6

Exploiting Structure in DisCSPs with Complex Local Problems

6.1 Introduction

Algorithms for solving DisCSPs generally assume, simplistically, that an agent represents a single variable and that they can be extended to handle multiple variables using "virtual" agents. In the previous chapter, we introduced strategies that combine heuristics in DisCSPs with one variable per agent. However, naturally distributed problems normally have multiple variables per agent called Complex Local Problems - (CLPs) hence, the problem structure could be exploited in solving them. In this chapter, we propose a novel divide and conquer approach for DisCSPs with CLPs that combines several search strategies. The approach prioritises the satisfaction of the more expensive (inter-agent constraints) part of the problem thus, solutions to the distributed part of the problem should be considered early in the search. To solve a problem, the divide and conquer approach combines the following search strategies: (i) systematic and local searches; (ii) centralised and distributed searches; and (iii) a modified compilation reformulation strategy. We present Divide and Conquer Algorithm for Agents with Complex Local Problems (Multi-DCA), an overall distributed local search algorithm that implements the divide and conquer approach.

This chapter is organized as follows. The divide and conquer approach for solving DisCSP with CLPs is discussed in Section 6.2. Divide and Conquer Algorithm for Agents with Complex Local Problems - Multi-DCA is presented in Section 6.3. In Section 6.4, we present the results of empirical evaluations of

Multi-DCA on benchmark problems. Finally, we discuss our contributions in Section 6.5 and summarise in Section 6.6 respectively.

6.2 Divide and Conquer Approach

6.2.1 Background

Recall that in a DisCSP with CLPs, variables involved in constraints with variables in another CLP are *external variables* whereas those which only have constraints with variables in their same CLP are *internal variables*. Constraints between variables in different CLPs are *inter-agent constraints* and those between variables in the same CLP are *intra-agent constraints*. DisCSP algorithms generally assume that each agent controls only a single variable and that they can be solved using "virtual" agents. To handle complex local problems, compilation and decomposition are two standard reformulations (Yokoo & Hirayama 2000) by which a DisCSP with CLPs can be transformed to give exactly one variable per agent. The reader is referred to Chapter 2, Section 2.2.3 for more discussion on the reformulation strategies. These reformulation strategies have been found to scale well on different problem settings (Burke & Brown 2006a). Compilation was modified to consider only externally relevant solutions and was found to be more appropriate as the size and complexity of each agent's CLP increases, with an overall small number of inter-agent constraints and an overall small domain size of the variables. On the other hand, the decomposition approach is better as the number of inter-agent constraints and the domain size of the variables increase, with an overall small problem size.

The CLP could be further exploited for additional opportunities to speed-up resolution. For example, in a meeting scheduling problem where it is more expensive to find solutions to inter-departmental meetings. Thus, to solve the inter-departmental meetings, consistent intra-departmental solutions to external meetings is found first and extended to solve the distributed problem. Each department then checks internally to satisfy constraints involving its local meetings. We propose a divide and conquer approach to solving DisCSP's with CLPs that combines the following strategies: (i) both systematic and local searches; (ii) both centralised and distributed searches; and (iii) a modified compilation reformulation strategy. The approach exploits concurrency and problem structure for a more effective resolution of DisCSPs. In the next section, we describe the approach.

Table 6.1: The divide and conquer approach

Stages	**External Variables**	**Internal Variables**
1	Each agent finds *compound groups* amongst its external variables. It divides them into smaller *compound groups* where required.	
2	For each *compound group*, a *centralised systematic search* finds its solutions. If no solution is found for the compound group, the problem is unsolvable.	
3	A *distributed local search* finds combinations of *compound group* solutions (found in Stage 2) which satisfy all inter-agent constraints.	
4		For each agent, a *centralised systematic search* extends the combination from the distributed local search (Stage 3) into a solution to its CLP. If all agents find an extension to the combination which is a solution to their CLP a solution is declared.

6.2.2 The Approach

In the divide and conquer approach, the structure of the DisCSP is analysed in order to identify groups of directly-related (via an intra-agent constraint) external variables within each agent. These groups are referred to as **compound groups**. A combination of local search and systematic search algorithms are then run concurrently to solve the divided problem. Firstly, each agent finds all locally consistent solutions to each *compound group*, using one systematic search per *compound group*. Secondly, using a distributed local search algorithm, the solutions found for each *compound group* are then combined with solutions to other agents' *compound groups* to form a **combination solution** such that all inter-agent constraints are satisfied. Finally, agents check if the compound solutions to its local compound groups participating in the combination solution extend to satisfy the rest of their intra-agent constraints to become solutions to their CLP.

The searches in the divide and conquer approach interleave, thus, exploiting concurrency. An overview of the divide and conquer approach is illustrated in Table 6.1. Next, we present the divide and conquer approach in Stages 1-4.

Stage 1: For each agent, all *compound groups* are found in a modified compilation reformulation strategy on the external variables. Unlike the basic compilation strategy where only one single complex variable is formed, several complex variables could be formed in this approach. The possible number of compound solution becomes exponential as the number of variables and domain size in a compound group increases. Thus, the size of the *compound groups* is restricted so that large *compound groups* are divided [1] into smaller *compound groups* (see Section 6.3.2 on determining group size).

Stage 2: For each *compound group*, a systematic search finds variable instantiations (solutions) which satisfy all the (intra-agent) constraints between its variables. Not all the variables within the agent are considered. The remaining variables are internal variables and are considered in Stage 4. As soon as one solution is found for each *compound group*, Stage 3 (see below) will commence. Meanwhile, Stage 2 will continue until all solutions to all *compound groups* are found, or the maximum number of iterations is reached in Stage 3 (see below) or a solution to the intra-agent constraints is found in Stage 4 (see below).

Stage 3: A distributed local search algorithm finds combinations of *compound group* solutions which satisfy all the inter-agent constraints. When a suitable *combination solution* is found, Stage 4 (see below) commences, but Stage 3 will continue the process of finding combination solutions until a maximum number of iterations is reached or a solution is declared in Stage 4. Note that each compound group forms a complex variable whose domain values are solutions found in Stage 2.

Stage 4: The *combination solution* found in Stage 3 is extended locally by each agent (i.e each agent considers the compound solutions to its local compound groups participating in the combination solutions) as well as the satisfaction of constraints between variables in different compound groups. Each agent ensures the satisfaction of the variables it represents (those not involved in a compound group). Each agent uses a systematic search algorithm to instantiate its other internal variables to satisfy the remaining intra-agent constraints (those not considered in Stage 2) into a complete solution to its CLP. If each agent can extend the partial solution for its variables from the combination solution into a CLP solution, a solution to the problem has been found and the search stops.

Table 6.2 summarises how a DisCSP is divided and solved in the divide and conquer approach. Next we illustrate the approach with an example.

[1]This does not apply to Multi-DynAPP, an implementation of the divide and conquer approach discussed in Appendix C

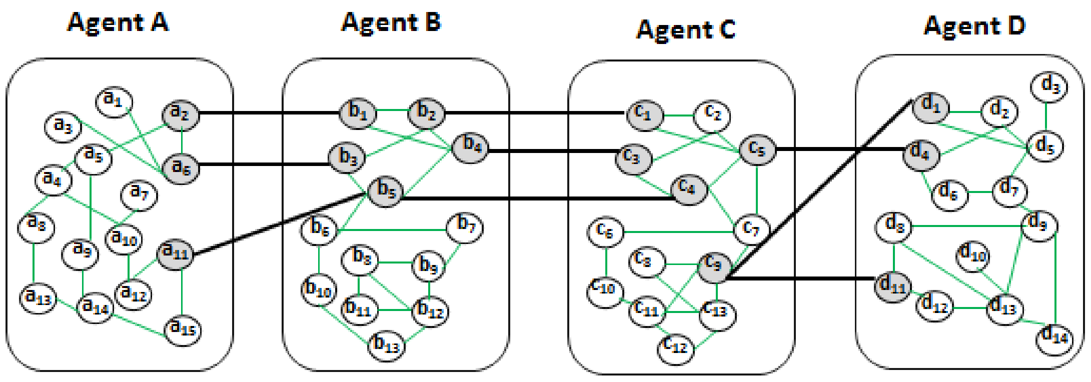


Figure 6.1: Example: DisCSP with CLPs

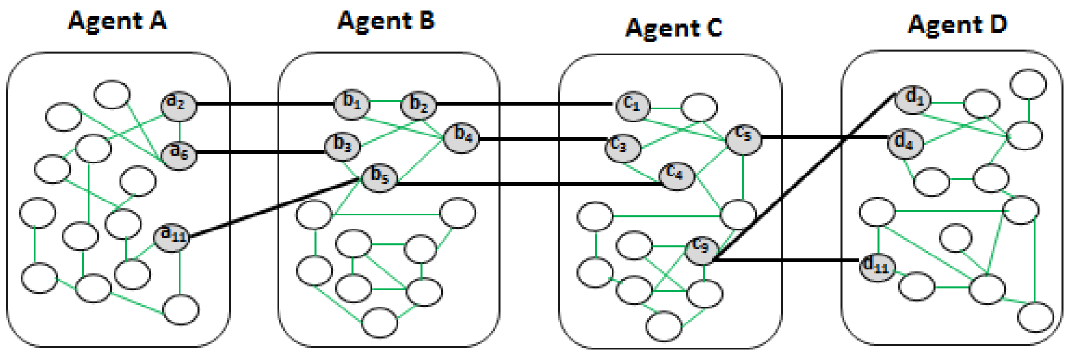


Figure 6.2: Example: external variables in a DisCSP with CLPs

6.2.3 Example

In this section, we use the example DisCSP with CLPs in Chapter 2, Section 2.2, presented again in Figure 6.1 to illustrate the concept of the divide and conquer approach. The figure illustrates a meeting scheduling problem as a DisCSP with 55 variables, 70 intra-agent constraints and 9 inter-agent constraints naturally distributed over four agents.

The external and internal variables of the example are presented in Figure 6.2 and Figure 6.3 respectively. The inter-agent constraints are represented with dark lines connecting variable belonging to different agents (see Figure 6.1). For example, the constraints between (a_2, b_1) and between (a_6, b_3). The intra-agent constraint are represented with green lines for example, the constraints between (a_2, a_6) and between (a_4, a_8).

When considering the structure of the DisCSP in Figure 6.1, external variables $\{a_2, a_6\}$ in Agent A, $\{b_1, b_2, b_3, b_4, b_5\}$ in Agent B and $\{c_1, c_3, c_4, c_5\}$ in Agent C each have a natural clustering that forms a

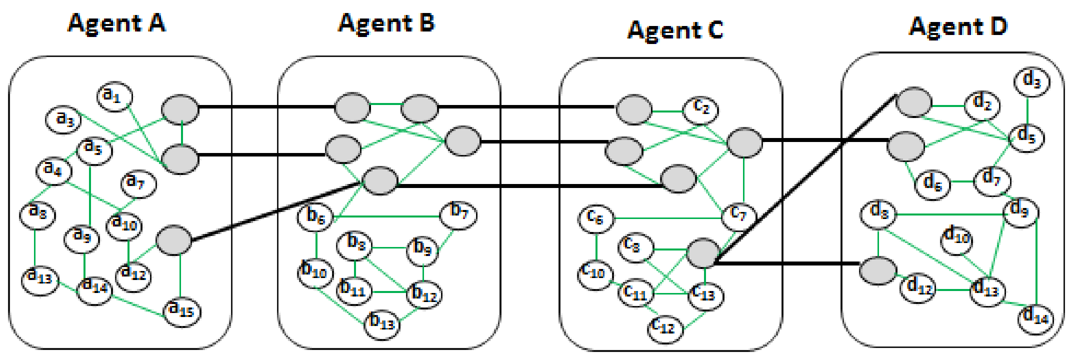


Figure 6.3: Example: internal variables in a DisCSP with CLPs

compound group because within each agent, the external variables share intra-agent constraints. However, other external variables such as a_{11} in Agent A, c_9 in Agent C and d_1, d_4, d_{11} in Agent D each form a compound group with a single external variable. Solving the distributed problem (i.e the inter-agent constraints) is more expensive compared to the local problem (i.e the intra-agent constraints) due to the number of messages sent among agents in negotiating a solution. To solve the DisCSP with the divide and conquer, solutions to the compound groups are first found which form *compound solutions*. For example, if each external variable in Agent A and Agent B has a domain size of 8 i.e $\in \{0, ..., 7\}$, an example domain of compound group [a_2, a_6] in Agent A would be {(1,1), (1,7), ..., (0,1)}, similarly, for compound group [b_1, $b_2$3, b_3, b_4, b_5] in Agent B would be {(1,1,1,1,1), (1,7,3,2,3), ..., (0,1,2,3,4)} and so on. Compound groups with large number of variables and domain size are divided to form smaller compound groups. Each agent then extends its compound solutions to form a combination solution that satisfies the inter-agent constraints. Finally each agent checks locally for assignments to the remaining variables in its CLP that extends the compound solution for a complete satisfaction of the DisCSP. A more detailed example of the approach is provided in Section 6.3.3. Next, we discuss an implementation of the divide and conquer approach.

6.3 Multi-DCA

Multi-DCA - Divide and Conquer Algorithm for Agents with Complex Local Problems is an implementation of the divide and conquer approach for solving DisCSP with CLPs. Multi-DCA uses a basic centralised systematic search algorithm (as described in 4.2.2) in Stage 2 and Stage 4. An adaptation of DynAPP for

Table 6.2: Multi-DCA: division of problem solving

	Centralised Systematic Search
Variables	External
Constraints	Intra-agent between external variables
Domain	Variable domain values
No. of runs	One per compound group in an agent
	Distributed Local Search
Variables	External
Constraints	Inter-agent
Domain	Solutions to *compound groups*
No. of runs	One involving all agent
	Centralised Systematic Search
Variables	Internal
Constraints	Intra-agent
Domain	Domain values
No. of runs	One per agent to check local consistency

compound domains (DynAPP-CD) is used as the distributed local search algorithm in Stage 3. Unlike DynAPP, DynAPP-CD does the following: (i) considers only inter-agent constraints; (ii) considers complex variables (within each agent, there could be several complex problems); (iii) uses compound values as domain (which are the solutions found in Stage 2 of the divide and conquer approach); and (iv) penalty messages are sent to all the neighbours. Algorithms 5 - 7 present the pseudocode for Multi-DCA.

6.3.1 Algorithm Details

In Algorithm 5, we present the Multi-DCA which starts by finding compound groups (see Algorithm 5, lines 5-7). The divided DisCSP is then solved using several search strategies (see Algorithm 5, lines 10-31). Next, we explain how the problem is divided then solved.

Finding Compound Groups In each agent, Algorithm 6 is used to identify *compound groups*. For each CLP, the *maxDegree* of each of the external variables is calculated (see Algorithm 6, line 6-8).

Algorithm 5 Multi-DCA: procedure **main**()

```
1: procedure $main$
2:     let $maxSteps$ be the given maximum number of iterations
3:     let A$_i$.Sol$_{comb}$ be the partial solution from Sol$_{comb}$ for variables in $A_i$, $i \in \{1..n\}$
4:     let $A_i.extlVars$ be external variables in $A_i$'s CLP
5:     end $= FALSE$, problemSolved $= FALSE$
6:     for each agent $A_i$, $i \in \{1..n\}$ concurrently do
7:         $A_i.CGs$ = $A_i$.findCompGroups($A_i.extlVars$)
8:     end for
9:     $CGs = \bigcup_{i=1}^{n} A_i.CGs$ {all compound groups}
10:    while not (end) concurrently do
11:        for each agent $A_i$ concurrently do
12:            for each $cg_k \in A_i.CGs$, k $\in$\{ 1..| $A_i.CGs$ |\} concurrently do
13:                Stage2($cg_k, end$);
14:            end for
15:        end for
16:        Stage3($CGs$,$problemSolved$,$maxSteps$);
17:        for each agent $A_i$ concurrently do
18:            if (A$_i$.Sol$_{comb}$ found in Stage3) then
19:                Stage4(A$_i$.Sol$_{comb}$,$problemSolved$);
20:            end if
21:        end for
22:        if ($\forall$ A$_i$, $i \in \{1..n\}$ extension found for A$_i$.Sol$_{comb}$) then
23:            Solution = $\bigcup_{i=1}^{n} A_i.Sol_{comb}\ extension$
24:            problemSolved $= TRUE$
25:            end $= TRUE$
26:        else
27:            if (Stage3 finished) then
28:                end $= TRUE$
29:            end if
30:        end if
31:    end while
32:    if (problemSolved) then
33:        print Solution
34:    else print "Solution not found"
35:    end if
36: end procedure
```

The variable with the largest maxDegree is used as a *seed* (start variable) to create a *compound group* (see Algorithm 6, line 9). Neighbours (i.e. external variables within the agent) of the seed are added to the *compound group* (see Algorithm 6, line 13-17) giving priority to neighbours with lower degrees (see Algorithm 6, line 14) until the maxGroupSize is reached. The variables in the new group are then removed from the available external variables (see Algorithm 6, line 12,16). If the variable has no available neighbours, it is returned as a compound group with one variable. This process is repeated by calculating the *maxDegree* of each of the remaining external variables to determine the seed for the next *compound group* until no more external variables need grouping. The *compound group(s)* are then returned (see Algorithm 6, line 21) and used in the **search algorithm** (see Algorithm 5, line 11-15) and described in Algorithm 7, Procedure Stage2.

The possible number of compound solutions becomes exponential as the number of variables in a compound group or as the domain size increases. In an empirical investigations on several group sizes and problem characteristic we found that a group consisting of 3 variables gave the optima results (refer to Section 6.3.2). Thus, the maximum size of a *compound group*, *maxGroupSize* is set to 3.

Search algorithms Agents in Multi-DCA perform several concurrent searches to solve the overall problem. The constraints in the DisCSP are classified into the following three types: (i) intra-agent constraints with variables in the same *compound group*; (ii) Intra-agent constraints with other variables in the same CLP, but which do not belong to the same *compound group*; and (iii) inter-agent constraints with variables in other agents. In Algorithm 7, the search algorithms at each stage consider some of the constraints (listed above).

1. In **Stage 2** (see Algorithm 7, Procedure Stage2, line 1-6), for each *compound group* a basic centralised systematic search (as described in 4.2.2) is used to find all solutions. If one of the *compound groups* in any of the agents has no solution, the problem is unsolvable (see Algorithm 7, Procedure Stage2, line 3-5).
2. In **Stage 3** (see Algorithm 7, Procedure Stage3, line 1-12), DynAPP-CD is used to combine solutions for *compound groups* into combination solutions which satisfy all inter-agent constraints. In this adaptation, penalty messages are sent to all neighbours.

Algorithm 6 Multi-DCA: findCompGroups(extlVars)

```
1: function findCompGroups(extlVars)
2:     let G be the set of compound groups, G = {}
3:     let $maxGroupSize$ be the max. no. of var. in a group
4:     while ($extlVars \neq \emptyset$) do
5:         for each $v \in extlVars$ do
6:             CalculateDegree()
7:         end for
8:         $H = \emptyset$    (create a new empty group)
9:         $v_{sel} = v \in extlVars$ with the largest degree
10:        let neighs$_{v_{sel}}$ be v's neighbours
11:        $H = \{v_{sel}\}$    (add variable to new group)
12:        neighs = neighs$_{v_{sel}} \backslash H$
13:        while (| H |< maxGroupSize and (neighs $\neq \emptyset$)) do
14:            $v_j$ = v $\in$ neighs with min degree
15:            $H = H \bigcup \{v_j\}$
16:            neighs = neighs $\backslash \{v_j\}$
17:        end while
18:        $G = G \bigcup \{H\}$    (add new group to G)
19:        $extlVars = extlVars \backslash G$
20:    end while
21:    return $G$
22: end function
```

Algorithm 7 Stages 2 to 4

```
1: procedure $Stage2(cg_k, end)$
2:     $sols_{cg_k}$ = solutions to $cg_k$ using exhaustive systematic search
3:     if ($sols_{cg_k} = \emptyset$) then
4:         end = TRUE
5:     end if
6: end procedure

1: procedure $Stage3(CGs, problemSolved, maxSteps, end)$
2:     for each agent $A_i$, $_i \in \{1..n\}$ concurrently do
3:         while $\exists cg_k \in A_i.CGs \mid sols_{cg_k} = \emptyset$ and not(end) do
4:             wait
5:         end while
6:     end for
7:     while (not (problemSolved) and not ($maxSteps$)) do
8:         for each agent $A_i$ , $i \in \{1..n\}$ concurrently do
9:             DynAPP-CD($A_i.CGs$)
10:        end for
11:    end while
12: end procedure

1: procedure $Stage4(Sol_{comb}, problemSolved, end)$
2:     Systematically extend $Sol_{comb}$ to satisfy CLP
3:     if ($no\ extensionfound$) then
4:         “Invalid solution”
5:     end if
6: end procedure
```

MaxSteps is the maximum number of iterations. While the given maxSteps is not reached (see Algorithm 7, Procedure Stage3, line 7-11), variables in *compound groups* must take values compatible with the compound solutions found in Stage 2, continuously finding several unique combination solutions until the maxSteps is reached or a solution to the DisCSP has been found. This stage passes the combination solutions found to Stage 4.

3. In **Stage 4** (see Algorithm 7, Procedure Stage4, line 1-6), a basic centralised systematic search (as described in 4.2.2) per agent is used in order to extend the compound solutions to its local compound groups participating in the combination solutions to become solutions to its CLP (see Algorithm 7, Procedure Stage4, line 2). Thus, instantiating its internal variables to satisfy the other intra-agent constraints not considered in Stage 2. If the solution cannot be extended by at least one agent, the combination solution is invalid (see Algorithm 7, Procedure Stage4, line 3-5).

If each agent finds a valid extension, an overall solution is found (see Algorithm 5, line 22-25) and it is returned (see Algorithm 5, line 33) otherwise a solution is not found (see Algorithm 5, line 34).

6.3.2 Determining the size of a group

The parameter needed to be determined in Multi-DCA is the maximum number of variables in a group for optimal performance. We performed a series of experiments on distributed graph colouring problems, random DisCSPs and distributed meeting scheduling problems using varying characteristics. For each problem class, we ran experiments with group size cutoff $\in \{2, 3, 4, 5, 6\}$. For each cutoff, we ran 100 runs of 100 solvable [2] problems. We measured the percentage of problem solved, number of non-concurrent constraint checks (NCCCs) and number of messages used by Multi-DCA. We then took a median of the 100 runs for each problem setting. The group size cut-off where the median number of messages and the median NCCCs is minimal is selected. This gives a measurement which indicates the optimal group size.

[2] We use only solvable problem for our evaluation because the algorithms we implement are distributed local search algorithms and they do not detect unsolvability

In Appendix C, we presented Multi-DynAPP [3] and found that there is a linear relationship between the size of the domain and the performance of the algorithm. Thus, to predict the relationship between the problem features; possible domain size and the optimal group size, we use linear regression to obtain Equation 6.1

$$cutoff = \lceil \alpha + (\beta * solutionSize) \rceil \tag{6.1}$$

where $solutionSize$ is Equation 6.2.

$$solutionSize = (D^{e}) \tag{6.2}$$

where $_e$ is the number of external variables and D is the domain size.

For each agent, we use a 70(30) internal(external) variable split. The problem parameters used are in the difficult region with the following characteristics;

(a) Distributed graph colouring problems with number of nodes n $\in$ {100, ..., 200} in steps of 20, colours $\in$ {3, 4, 5} and degree $\in$ {4.6, ..., 5.2} in steps of 0.1. We use 10 agents i.e., the number of variables per agent $\in$ {10, ..., 20} in steps of 1 with respect to the node size. With the 70(30) internal(external) variable split i.e external variables $\in$ {3, 3, 4, 5, 6} with respect to the number of variables. For example, for 150 variables and 10 agents, there are 15 variables per agent. From the 15 variables, 30% (i.e. 5 variables) are external variables.

(b) Random DisCSPs with number of variables n $\in$ {100, ..., 200} in steps of 10, each with 10 agents i.e. number of variables per agent $\in$ {10, ..., 20} in steps of 1. The domain size d $\in$ {7, 8, 9, 10}, constraint density $p_1 \in$ {0.1, 0.15, 0.2} and constraint tightness p_2 of 0.5.

(c) Distributed meeting scheduling problems with number of variables n $\in$ {100, ..., 200} in steps of 10, each with 10 agents i.e. number of variables per agent $\in$ {10,..., 20}. Time slots $\in$ {5, 6, 7}, constraint density $p_1 \in$ {0.1, 0.12, 0.14, 0.16, 0.18, 0.2} and maximum travel time $\in$ {2, 3}.

Note that our evaluation is on problems with the parameters listed above. We perform linear regression on the results and found values for α and β as presented in Table 6.4. For example, consider the sample

[3]a local search algorithm for solving DisCSPs with CLPs that implements a divide and conquer approach, unlike Multi-DCA, large compound groups are not partitioned

	Graph Colouring Problems					
Var	**Agents**	**DomainSize**	**NCCCs**	**Var/agent**	**Ext. Variables**	**SolSize**
	NCCCs					
100	10	3	20,809	10	3	27
120	10	3	59,334	12	4	81
140	10	3	85,245	14	5	243
160	10	3	95,750	16	5	243
	Messages					
100	10	3	50	10	3	27
120	10	3	60	12	4	81
140	10	3	50	14	5	243
160	10	3	130	16	5	243

	Random DisCSPs					
Var	**Agents**	**DomainSize**	**NCCCs**	**Var/agent**	**Ext. Variables**	**SolSize**
	NCCCs					
100	10	8	15,721	10	3	512
120	10	8	49,076	12	4	4,096
140	10	8	34,477	14	5	32768
160	10	8	96,306	16	5	32,768
	Messages					
100	10	8	481	10	3	512
120	10	8	570	12	4	4,096
140	10	8	1,055	14	5	32,768
160	10	8	1,326	16	5	32,768

	Meeting Problems					
Var	**Agents**	**DomainSize**	**NCCCs**	**Var/agent**	**Ext. Variables**	**SolSize**
	NCCCs					
100	10	6	18,914	10	3	216
120	10	6	36,910	12	4	1,296
140	10	6	41,558	14	5	7,776
160	10	6	53,795	16	5	7,776
	Messages					
100	10	6	50	10	3	216
120	10	6	65	12	4	1,296
140	10	6	70	14	5	7,776
160	10	6	80	16	5	7,776

Table 6.3: Multi-DCA: maximum group size cut-off: sample results

data in the random DisCSP in Table 6.3, having 120 variables, 10 agents and a domain size of 8. Each agent has 20 variables in its CLP. A 70(30) internal(external) variable split implies 4 external variables. The possible solution size is 8^4 which is 4096 and the cutoff = $\lceil(2.4 + (-0.0000041) * 4096)\rceil$ which equals a cutoff maximum group size of 3. Similarly, from the experiment conducted on distributed graph colouring and distributed meeting scheduling we derived the value formula for α and β and a cutoff of approximately group size of 3 gave the best results.

	Graph problem	Random problem	Meeting problem
α	2.2	2.4	2.4
β	0.00058	- 0.0000041	0.0000365

Table 6.4: Multi-DCA values for α and β

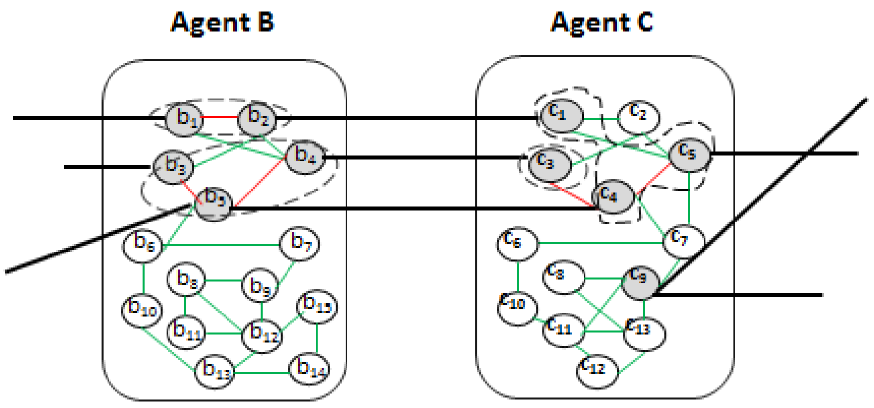


Figure 6.4: Multi-DCA: dividing compound groups

6.3.3 Example of Multi-DCA Execution

In this section, we use the example DisCSP with CLPs presented in Figure 6.1 to demonstrate the execution of Multi-DCA.

Finding compound groups Agent A has 3 external variables in 2 groups consisting of [a_2, a_6], [a_{11}] which are not further divided. Agent B has 5 external variables [b_1, b_2, b_3, b_4, b_5] and is further divided (see Figure 6.4). First, the degree of each variable is calculated and the variable with the largest degree is used as seed to create a *compound group*. The degrees of variables are as follows: the degree of variable b_1 is 3, variable b_2 is 4, variable b_3 is 3 variable b_4 is 4 and variable b_5 is 5.

A *compound group* is created with variable b_5 as seed and its neighbours variables b_3 and b_4 to form [b_3, b_4, b_5] (see Figure 6.4). These are removed from the available set of variables to group, leaving variables b_1 and b_2. Next, variable b_2 has the highest degree so it becomes the seed for a new compound group. Variable b_1 is related to it, so it is selected to go in the same *compound group* [b_1, b_2]. Agent C has 5 external variables in 2 groups consisting of [c_1, c_3, c_4, c_5] and [c_9].

The group [c_1, c_3, c_4, c_5] needs to be divided. The degrees of the variables are as follows: the degree of variable c_1 is 3, variable c_3 is 3, variable c_4 is 4 and variable c_5 is 5. A *compound group* is created with

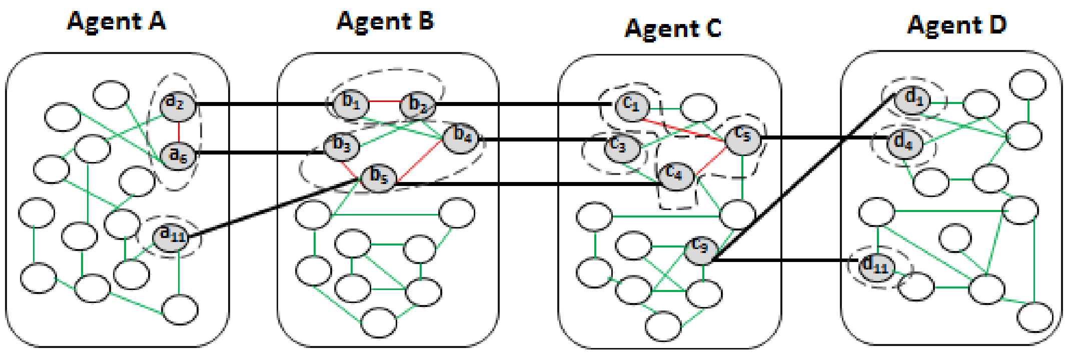


Figure 6.5: Multi-DCA: intra-agent constraints in compound Groups

variable c_5 as seed and its neighbours variables c_1 and c_4 to form [c_1, c_4, c_5] and [c_3] becomes a single variable group (see Figure 6.4). Agent D has 3 external variables in 3 groups consisting of [d_1], [d_4], [d_{11}] and is not divided.

Search algorithms

1. In Stage 2, for each *compound group* a basic centralised systematic search (as described in 4.2.2) is used to find consistent solutions. In our example, to satisfy the constraints shown in *red* in Figure 6.5, Agent A instantiates variables in the *compound group* [a_2, a_6], a_{11} forms a compound group consisting of a single variable and thus it maintains its default domain. In Agent B there are two searches, one to instantiate variables in the *compound group* [b_1, b_2] and another to instantiate the variables in the *compound group* [b_3, b_4, b_5]. Agent C has one search, to instantiate variables in the *compound group* [c_1, c_4, c_5]. The reminder compound group in Agents C and D contain only one variable and, therefore the compound domains of those are the domains of their variables.

2. In Stage 3, an adaptation of DynAPP (DynAPP-CD) is used to combine solutions for *compound groups* into combination solutions which satisfy all inter-agent constraints. In this adaptation, variables in *compound groups* must take values compatible with the compound group solutions found in Stage 2.

   Referring back to our example in Figure 6.4, the compound solutions and single group domain values (i.e. [a_2, a_6], [a_{11}]) in Agent A, ([b_1, b_2], [b_3, b_4, b_5]) in Agent B, ([c_1], [c_1, c_4, c_5]) Agent C and ([d_1], [d_4], [d_{11}] in Agent D are considered by the respective agents to combine the solutions with the compound solutions of their neighbour agents to find *combination solutions* which satisfy the inter-agent constraints (shown in *bold*). For example, Agent A participates in a search which combines compound solutions to [a_2, a_6], [a_{11}] with compound solutions to ([b_1, b_2], [b_3, b_4, b_5]). The *combination solutions* are used in Stage 4.

3. In Stage 4, a basic systematic search (as described in 4.2.2) per agent is used in order to extend the partial combination solutions relevant to each agent to be solutions to its CLP. In the example in Figure 6.4 Each Agent extends their relevant part of the partial consistent *combination solutions* by instantiating the rest of its internal variables to solve the remaining intra-agent constraints not considered in Stage 2 (i.e. the constraints shown in *green*).

6.3.4 Multi-DCA Algorithmic Properties

Termination: Multi-DCA will terminate in one of the following situations;

- Stage 2: If an agent declares a *compound group* unsolvable using systematic search (Algorithm 7, Procedure Stage2, lines 4).
- Stage 3: If the distributed incomplete solver reaches the maximum number of iterations and no solution has been found (Algorithm 7, Procedure Stage3, line 11-13).
- Stage 4: Two ways of terminating. (i) If Stage 2 within an agent has finished and returned some solutions but none of the solutions could be extended to a complete solution to the agents' CLP (Algorithm 7, Procedure Stage4, line 9-12); and (ii) If an overall solution is found (Algorithm 7, Procedure Stage4, line 14-16).

Completeness: Multi-DCA uses a distributed local search algorithm to find consistent compound solutions. Consequently, Multi-DCA is incomplete thus, it cannot prove the unsolvability of a problem and it may fail to find a solution to a problem when a solution exists. However, in Stage 2, unsolvability can be detected when no solution is found for any of the compound groups (which are solved with an exhaustive systematic search).

Privacy : In Multi-DCA, agents only exchange partial assignments with their neighbours. The complete details of a CLP are only know by the agent that represents it. In each stage of Multi-DCA, privacy is maintained as follows;

- In Stage 2: Only a subset of the centralized problem is considered.
- In Stage 3: Partial solutions found in Stage 2 above are used to solve the distributed problem. Each agent only knows variables, the current assignment of the variables represented by its neighbour agents together with the constraints relating them. It does not know other variables represented by the neighbours or their other constraints
- In Stage 4: Each agent receives partial consistent assignment of its variables then independently extends the partial solution by instantiating its internal variables and checking its other intra-agent constraints not yet considered to determine complete consistence to its CLP.

6.4 Empirical Evaluation of Multi-DCA

In this section, we discuss the performance of Multi-DCA on a number of solvable [4] problems; distributed graph colouring problems, random DisCSPs, distributed meeting scheduling problems and distributed sensor network (refer to Chapter 3 for the description of problems). For each problem setting, we run 100 solvable problem instances from which we record and present: (i) the percentage of problems solved within the given maximum number of iterations; (ii) median of the number of Non-Concurrent Constraint Checks (NCCCs) performed; and (iii) median of the number of messages sent to measure efficiency. Although CPU time is not an established measure for DisCSPs (Meisels et al. 2002) and is not reported in this thesis, we measured CPU time and found the results to be consistent with the other evaluation metrics used. To generate naturally distributed DisCSPs (except for distributed sensor network problems) with complex local problems (i.e. with a higher proportion of intra(inter)agent constraints), the problems considered contained between 70(30) and 80(20) ratio of intra(inter)agent constraints and the exact ratio used in the result presented is specified in the experiments. The ratio of variables within the CLP are 70(30) internal(external) variables. Each algorithm was allowed to run for a maximum of (100 * (n)) iterations where n represents the number of variables (nodes, variables, meetings or sensors).

The results are benchmarked with three other DisCSP algorithms that handle CLPs: **Multi-DisPeL**, **DisBO-wd** and **Multi-Hyb-Pen**. Multi-DisPeL (value penalty) and DisBO-wd (constraint weights) are local search algorithms while Multi-Hyb-Pen is a hybrid (complete) search algorithm (refer to Chapter 4,

[4] We use only solvable problem for our evaluation because the algorithms we implement are distributed local search algorithms and do not detect unsolvability

Section 4.3 for the description of algorithms). We obtained the implementations of Multi-DisPeL, DisBO-wd and Multi-Hyb-Pen from their authors.

Note: The following algorithms were not considered; (i) DCDCOP (Khanna et al. 2009) was not considered for evaluation as the pseudocode available is insufficient to implement the algorithm and it is designed for DCOP; (ii) Burke's work (Burke & Brown 2006a), (Burke & Brown 2006b) concentrated on efficiency in handling CLPs and there is no overall algorithm; and (iii) ADOPT (Modi et al. 2005) is also designed for distributed constraint optimization. In the few cases where not all problems were solved, the effort "wasted" in that problem are not counted towards the median number of messages and median NCCCs, they do however affect the percentage of problems solved. Multi-Hyb-Pen is a complete algorithm and it solved all problems as expected. Because Multi-DCA could be terminated at other stages and not only Stage 4 (distributed local search), we did not perform a run length distribution.

6.4.1 Results on Distributed Graph Colouring Problems

We use 3-colour distributed graph colouring problems with different characteristics to determine the performance of Multi-DCA on several settings. We present results for experiments where we vary the following problem parameters: (i) degree; (ii) number of nodes; (iii) number of agents; and (iv) domain size.

- **Varying Degree**: For these experiments, we use a problem with 300 nodes and vary the degree $\in \{4.4, ..., 5.1\}$ in steps of 0.1. The results are presented in Figure 6.6. Multi-DCA solved more problems compared to both Multi-DisPeL and DisBO-wd.

  On NCCCs, Multi-DCA used the least NCCCs around degree deg $\in \{4.8,..., 5.1\}$ (which includes the region of difficult problems with deg at 4.9). The performance observed for Multi-DCA on number of messages is similar to the results of NCCCs at degree $\in \{4.9, 5.0\}$, Multi-DCA exchanged the least number of messages. But on the other degrees, i.e where the problems are easier, Multi-DCA did not do well on number of messages exchanged.

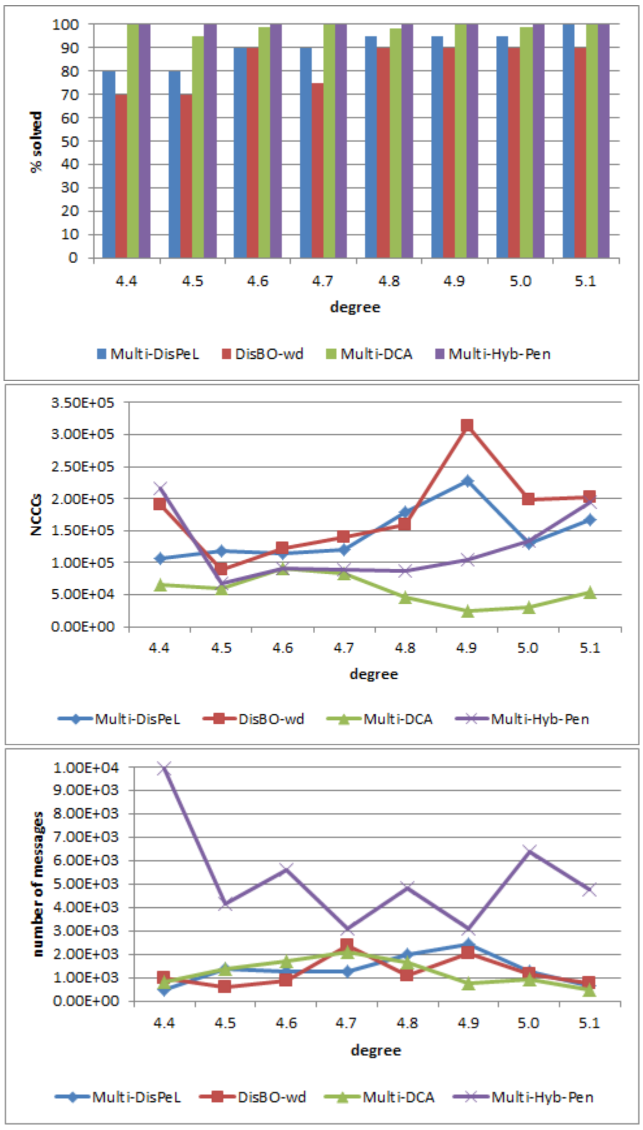


Figure 6.6: $< n = 300, |colours| = 3, |agents| = 15, deg \in \{4.4, ..., 5.1\} >$

- **Varying the Number of Nodes:** For these experiments, we use number of nodes n ∈ {200, ..., 300} in steps of 20 and fixed the degree at 5.0. We then conduct two sets of experiments by using a (i) fixed number of nodes; or (ii) varying number of nodes within an agent. This is to determine the performance of increasing number of nodes on agents, by sharing the problem on a fixed or varying number of agents.

  (i) The number of nodes per agent varies for each problem

  The results are presented in Figure 6.7. All algorithms solved all problems except at n ∈ {200, 220, 240} where DisBO-wd solved 90%, 90% 80% respectively. Initially, for n ∈ {200, 220}, Multi-Hyb-Pen had the least NCCCs and as the number of nodes increases, Multi-DCA did better by having the least NCCCs. On number of messages, Multi-DCA generally exchanged the least number of messages.

  (ii) The number of nodes per agent is fixed for each problem

  The number of nodes within an agent is fixed to 10 by increasing the number of agents. Thus, as the number of nodes in the DisCSP is increased, more agents are used. For example, for 200 nodes, we use 20 agents, for 220 variables we use 22 agents and so on. The results are presented in Figure 6.8. All the algorithms solved all problems and thus, we exclude % of problems solved from our illustration. Agents in Multi-DCA consistently used the least number of messages and had the least NCCCs.

  For both fixed and varying number of agents, we observe that, as the number of nodes increases, from 220 nodes to 300 nodes, Multi-DCA generally gives the overall best results on NCCCs and number of messages.

- **Varying the Number of Agents:** In these experiments, we fixed the number of nodes n at 300. We use a degree of 5.0 with a varying number of agents, thus, results in varying the number of variables belonging to an agent. We use agents ∈ {10, 15, 20}. The results are presented in Figure 6.9. From these results, all the algorithms solved the entire problems except DisBO-wd which solved 95% of the problems when 15 agents were used. On NCCCs and number of messages, Multi-DCA generally gave the overall best. The number of messages increased for all algorithms at 20 agents. Since each agent has less knowledge about the problem, more communication is required.

- **Varying the Number of Colours:** We vary the domain size (number of colours) ∈ {2, 3, 4, 5}

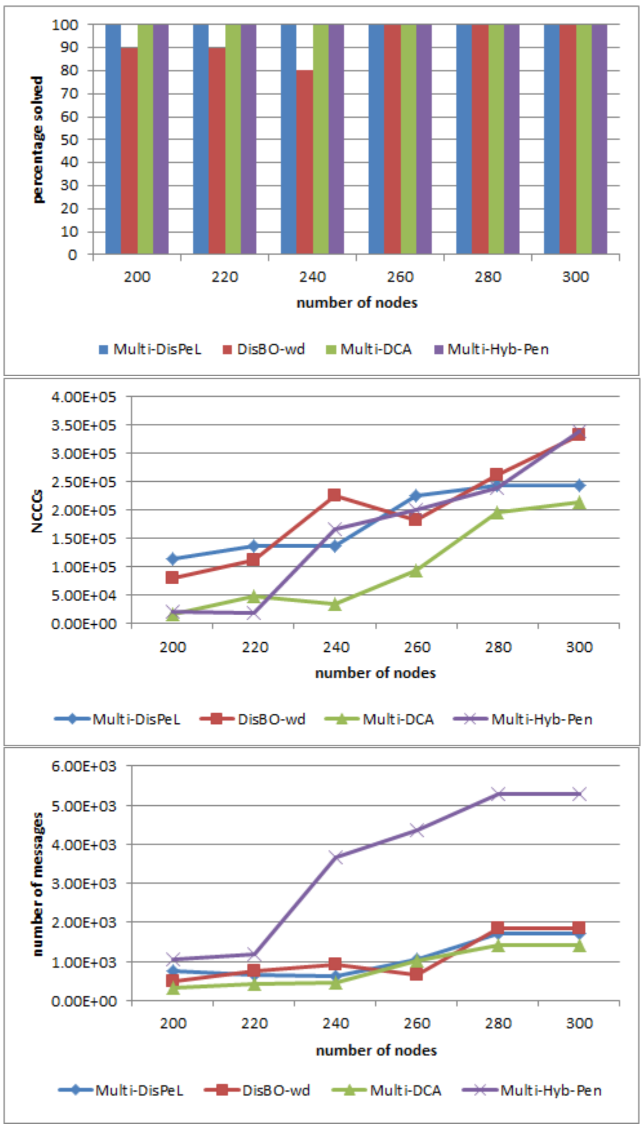


Figure 6.7: < n ∈ {200, ..., 300}, |colours| = 3, |agents| = 10, deg = 5.0>

in this experiments and fixed the number of nodes at 300, degree at 5.0 and use 10 agents. The results are presented in Figure 6.10. All the algorithms solved all problems except DisBO-wd which solved 97% and 98% of the problems when 2 or 3 colours were used respectively. On both NCCCs and number of messages, Multi-DCA, Multi-DisPeL and DisBO-wd used less than Multi-Hyb-Pen. Multi-DCA generally performed best on NCCCs and number of messages.

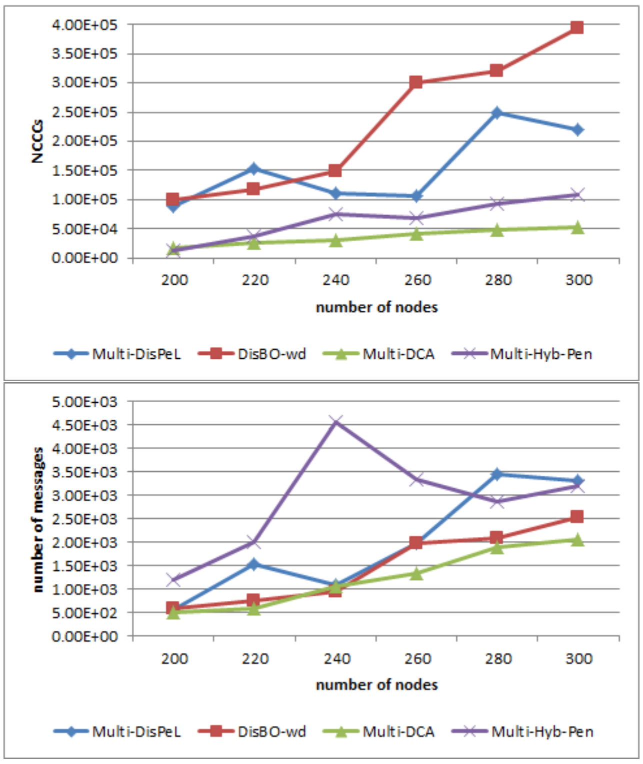


Figure 6.8: $< n \in \{200, ..., 300\}, |colours| = 3, |agents| \in \{20, ..., 30\}, deg = 5.0 >$

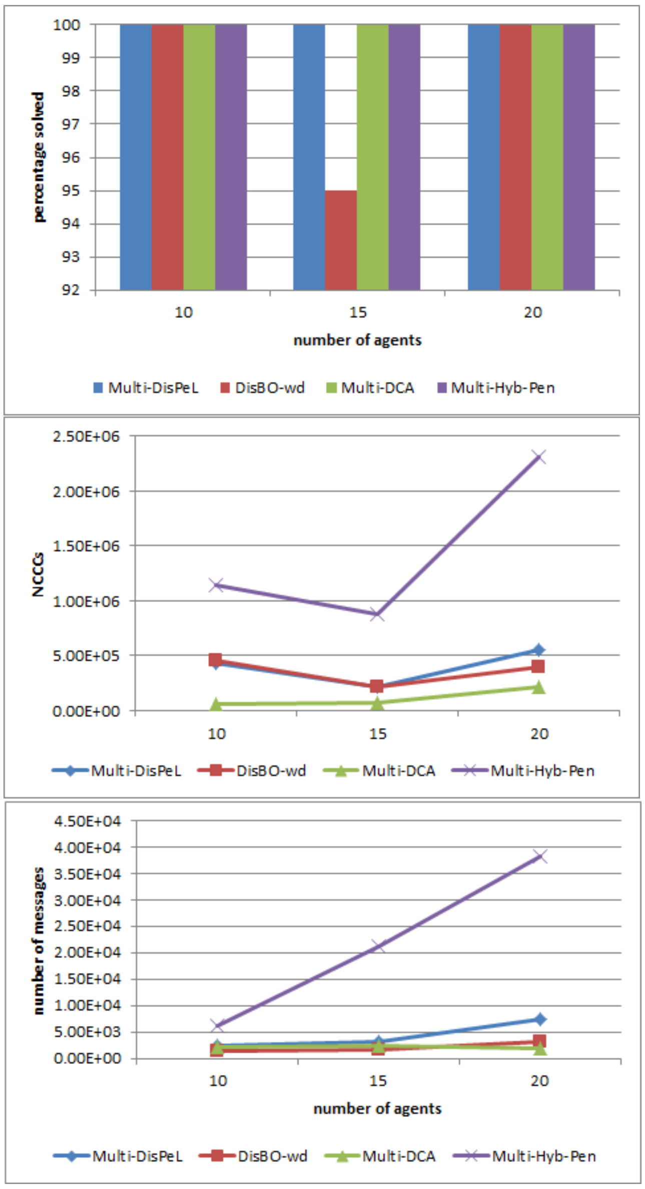


Figure 6.9: $< n = 300, |\text{colours}| = 3, |\text{agents}| \in \{10, 15, 20\}, \text{deg} = 5.0 >$

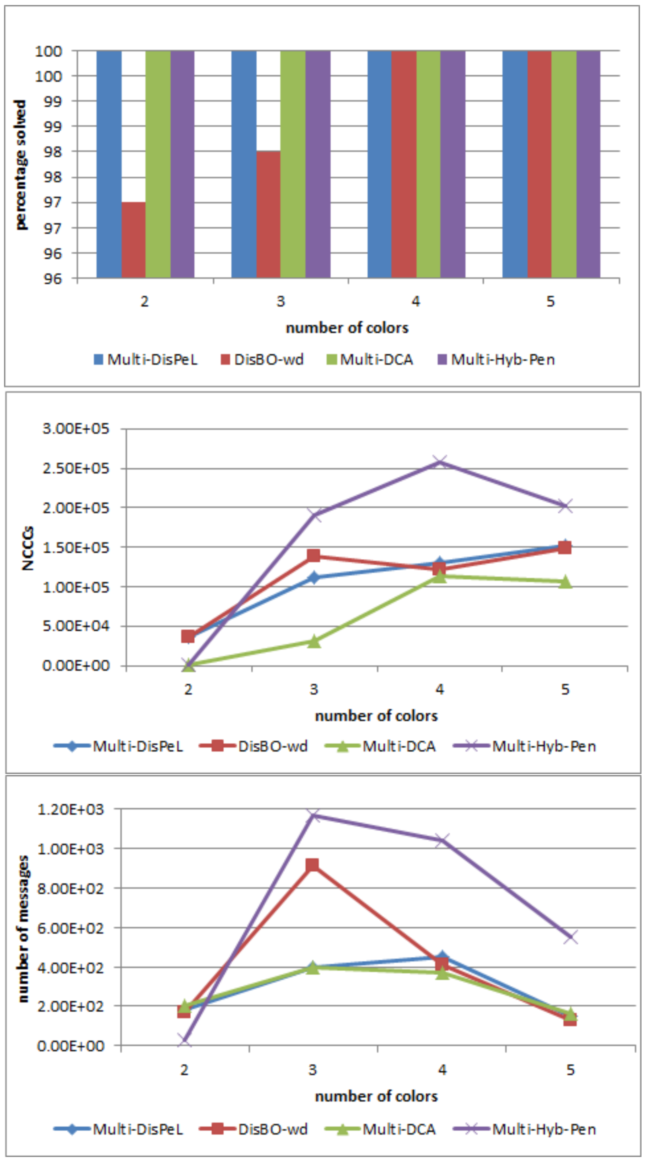


Figure 6.10: $< \text{n} = 300, |\text{colours}| \in \{2, 3, 4, 5\}, |\text{agents}| = 15, \text{deg} = 5.0 >$

6.4.2 Results on Random Distributed Constraint Satisfaction Problems

On random DisCSP, we present results for experiments where we vary the following problem parameters: (i) constraint tightness; (ii) number of variables; (iii) number of agents; and (iv) domain size. We use a ratio of 70(30) intra (inter) agent constraints.

- **Varying Constraint Tightness:** We generate 100 instances of a problem with 150 nodes, a domain size of 8, constraint density p_1 of 0.5 and constraint tightness $p_2 \in \{0.3, ..., 0.7\}$. All four algorithms solved all problems. The results are presented in Figure 6.11. All algorithms solved all the problems and this is not illustrated. For NCCCs, Multi-DCA outperformed Multi-Hyb-Pen, Multi-DisPeL and DisBo-wd at p_2 = 0.3 and 0.4 i.e at the region of difficult problems for NCCCs. Multi-DCA did not do well for number of messages exchanged.

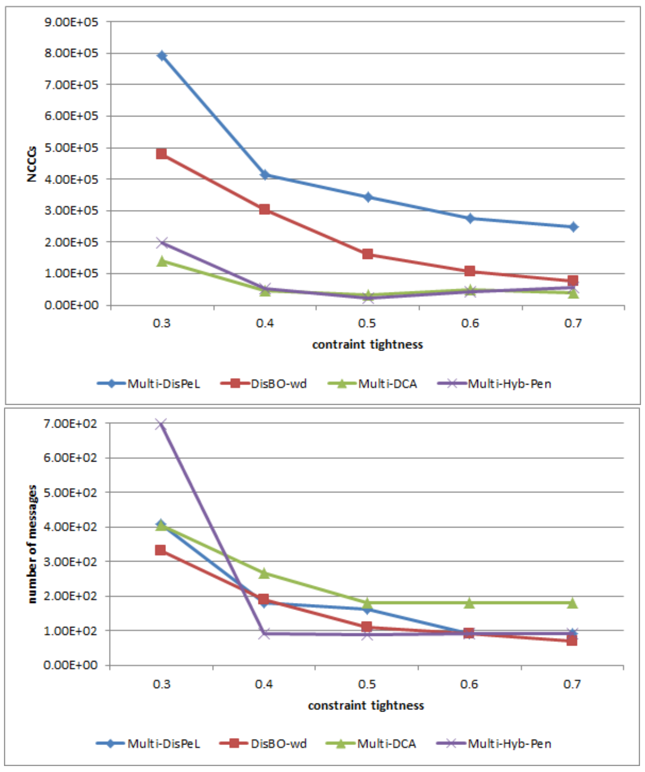


Figure 6.11: $< n = 150, |domain| = 8, |agents| = 10, p_2 \in \{0.3, ..., 0.7\}, p_1 = 0.15 >$

- **Varying the Number of Variables** : In these set of experiments, we generate problems of different sizes i.e. number of variables $n \in \{160, 170, ..., 250\}$ in steps of 10, with a constraint density p_1 of 0.2. We use this setting to conduct two sets of experiments by using (i) a fixed number of variables; or (ii) a varying number of variables within an agent. This is to determine the performance of increasing number of variables on agents, by sharing the problem on a fixed or varying number of agents.

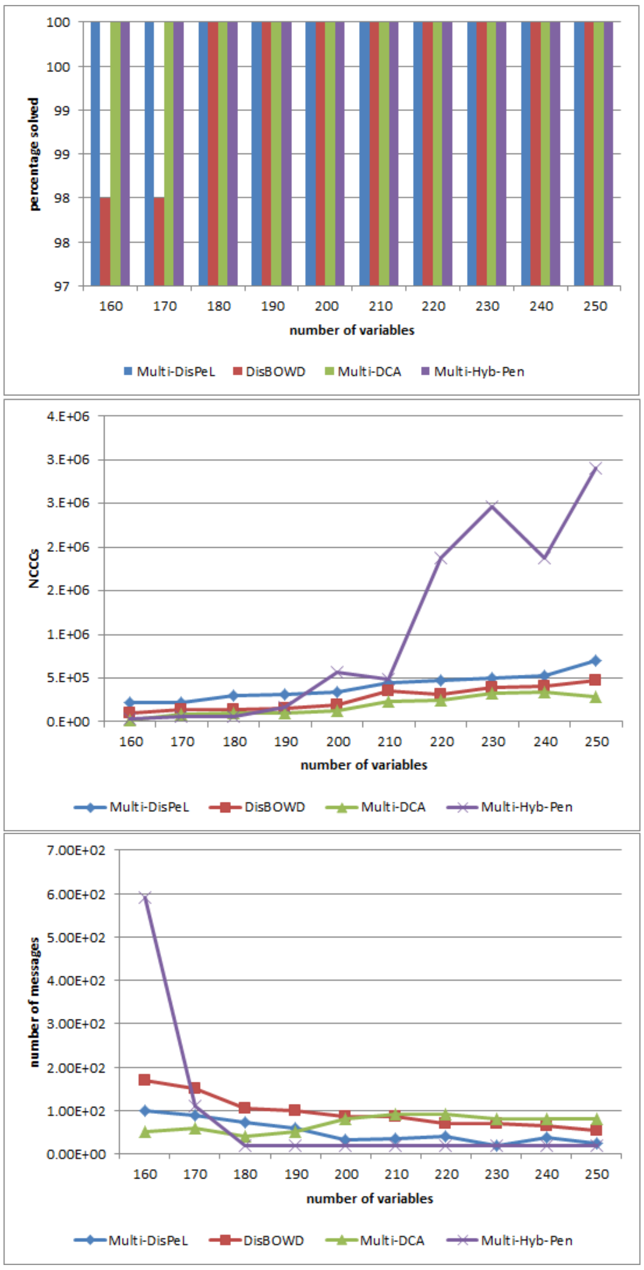


Figure 6.12: $< n \in \{160, ..., 250\}, |\text{domain}| = 8, |\text{agents}| = 10, p_2 = 0.4, p_1 = 0.2 >$

1. **The number of variables per agent varies for each problem**

   We use 10 agents for each problem size thus, as the problem sizes increases, the number of variables within an agent also increases. For example, for 110 variables there would be 11 variables/agent, for 120 variables, there would be 12 variables/agent and so on. The results are presented in Figure 6.12. Multi-DisPeL, DisBO-wd, Multi-DCA and Multi-Hyb-Pen generally solved all the problems except for variables $\in \{160, 170\}$ where DisBO-wd solved 98% of the problems. Multi-DCA generally performed better than all the algorithms by having the least NCCCs as the number of variables increased. However on Multi-DCA did not do well for number of messages exchanged.

2. **The number of variables per agent is fixed for each problem**

   We fixed the number of variables in an agent to 10 by varying the number of agents. The results are presented in Figure 6.13.

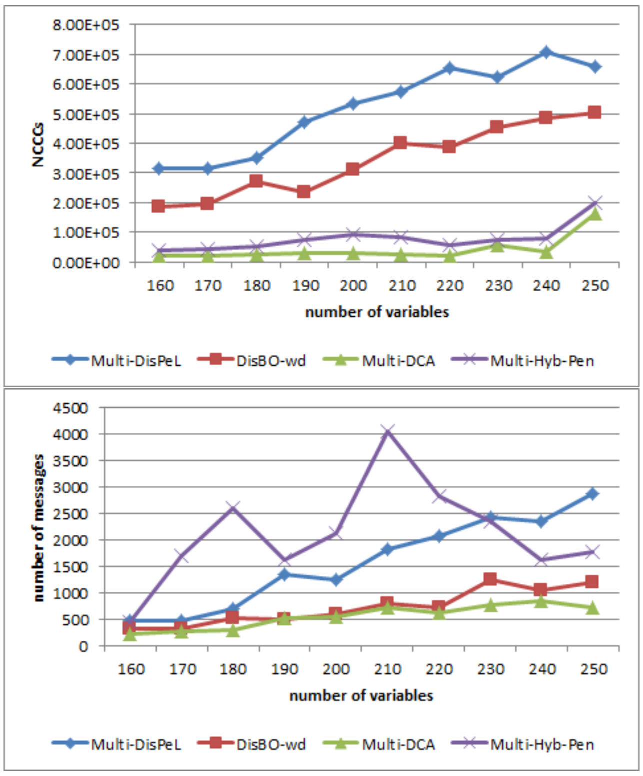
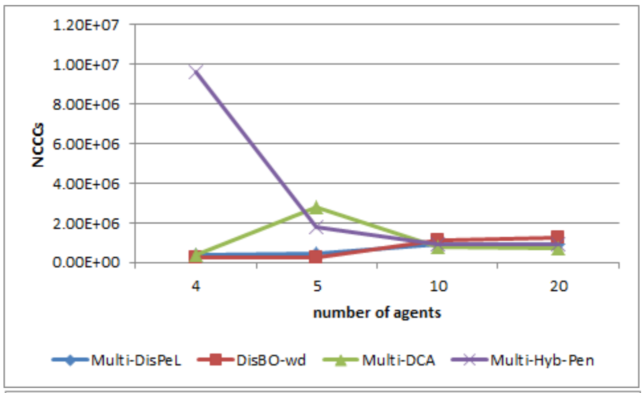


Figure 6.13: $< n \in \{160, ..., 250\}, |domain| = 8, |agents| \in \{16, ..., 25\}, p_2 = 0.4, p_1 = 0.2 >$

Multi-DisPeL, DisBO-wd, Multi-DCA and Multi-Hyb-Pen solved all the problems and this is not illustrated. For NCCC, Multi-DCA had the least compared to Multi-Hyb-Pen, Multi-DisPeL and DisBO-wd. Similarly on messages, Multi-DCA generally used the least number of messages compared to all the other algorithms.

- **Varying the Number of Agents:** In this experiments, we used a fixed number of variables 200 and a tightness p_2 of 0.4. However, we vary the number of agents $\in \{4, 5, 10, 20\}$, i.e. the number of variables within an agent becomes $\{50, 40, 20, 10\}$ with respect to the number of agents. The results are presented in Figure 6.14. We observe that, the performance of Multi-DCA was best as the number of agents increased i.e. at agent $\in \{10, 20\}$, using the least number of messages and NCCCs. All algorithms used the least number of messages when 4 agents where used as each agent has more knowledge about the problem, less communication between agents is required.



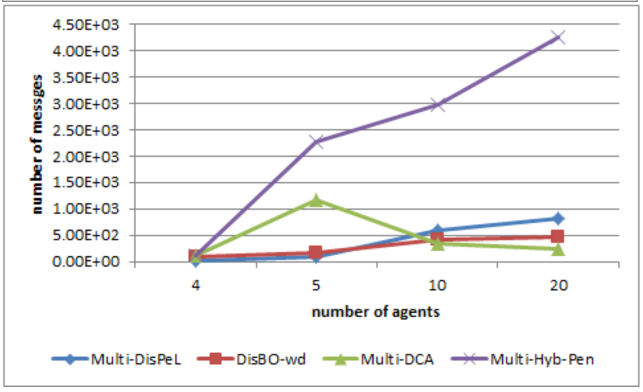


Figure 6.14: $< n = 200$, $|\text{domain}| = 8$, $|\text{agents}| \in \{4, 5, 10, 20\}$, $p_1 = 0.15$, $p_2 = 0.4 >$

- **Varying the Domain Size:** We use random DisCSPs at the region of hard problems (i.e. phase transition) with a constraint tightness p_2 of 0.5, a variables size of 150 and 10 agents. We vary the domain $d \in \{3, ..., 10\}$ in steps of 1. All the solvers solved all the problems. The results are presented in Figure 6.15. Multi-DCA generally had the least NCCCs as the size of the domain increases. However, on the number of messages exchanged, Multi-DCA didn't do well.

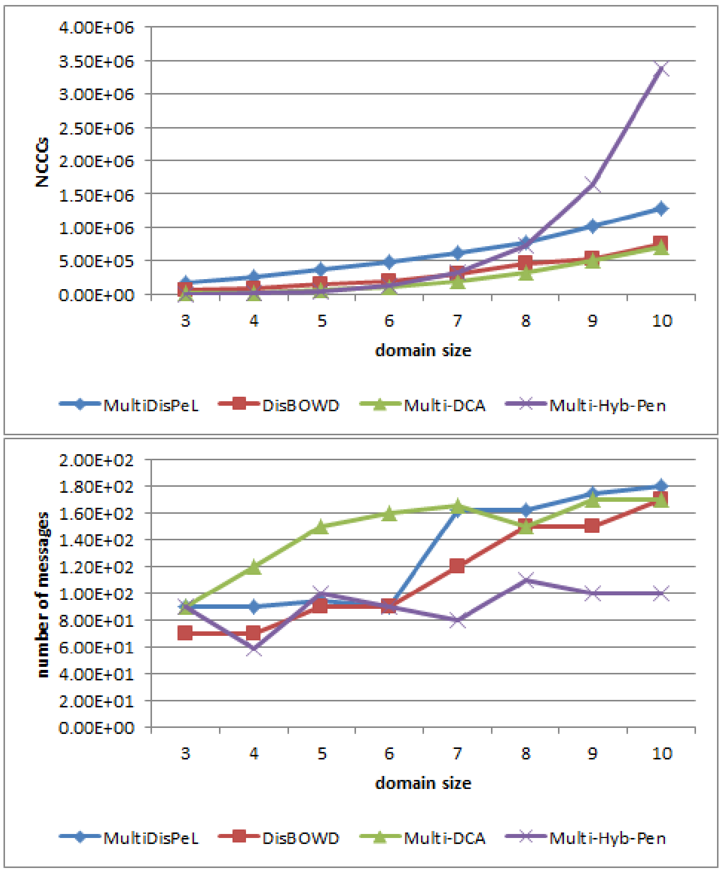


Figure 6.15: $< n = 150, |\text{domain}| \in \{3, 4, ..., 10\}, |\text{agents}| = 10, p_1 = 0.1, p_2 = 0.4 >$

6.4.3 Results on Distributed Meeting Scheduling Problems

To evaluate Multi-DCA on meeting scheduling problems, we use problems with several meetings and constraint density. We present results for experiments where we vary the following problem parameters: (i) timeslots; (ii) constraint density; and (iii) number of meetings. We use a ratio of 70(30) intra (inter)agent constraints.

- **Varying the Timeslots**: We use 200 meetings with timeslots $\in \{4, ..., 8\}$ in steps of 1. We use 5 agents for each problem size, a constraint density d of 0.2 and a maximum travel time md of 3. The results are presented in Figure 6.16. Multi-DCA solved atleast as much problems as all the local search algorithms Multi-DisPeL and DisBO-wd. The NCCCs and number of messages exchanged generally increased as the number of timeslots increases and Multi-DCA performed closely to Multi-

HybPen.

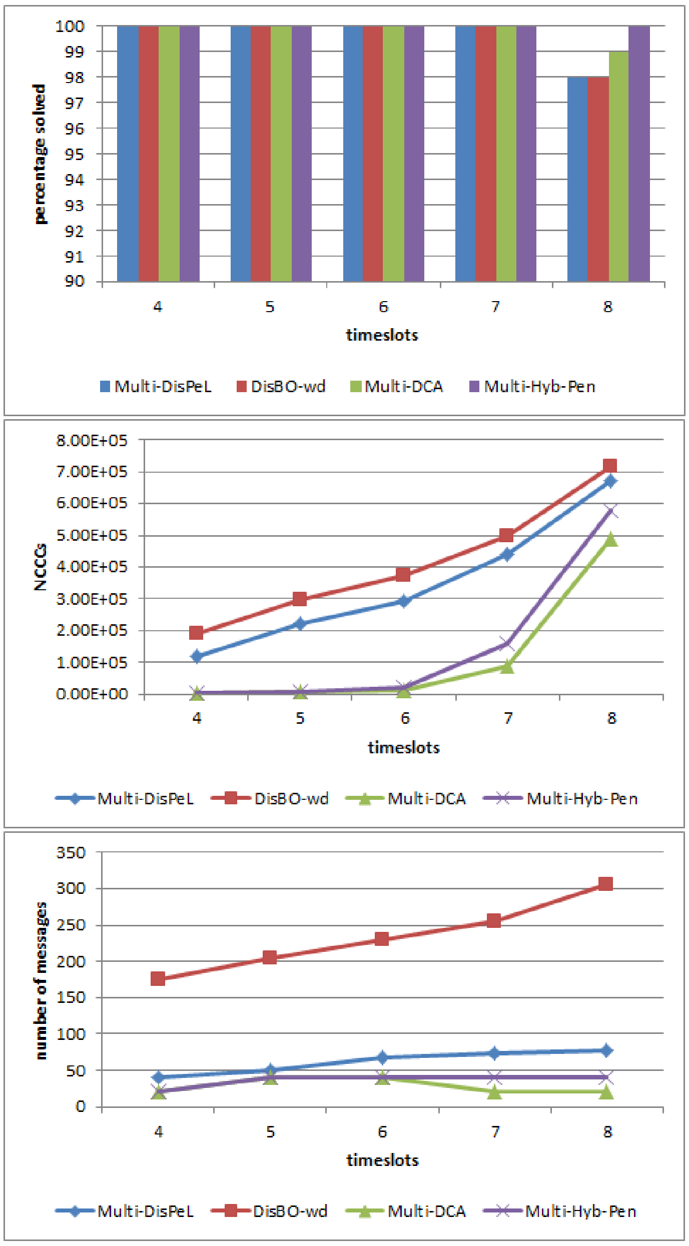


Figure 6.16: < n = 200, timeslots ∈ {4, 5, ..., 8}, |agents| = 10, d = 0.2, md = 3 >

- **Varying the Constraint Density**: We use 200 meetings with 5 agents, a timeslot of 6, a constraint density d ∈ {0.1, ..., 0.22} in steps of 0.02 and a maximum travel time md of 3.

  The results are presented in Figure 6.17.

  Multi-DCA solved at least as much as the local search algorithms Multi-DisPeL and DisBO-wd. For NCCC, at constraint density 0.1 and 0.12, Multi-DCA had the mot NCCCs but for the rest of

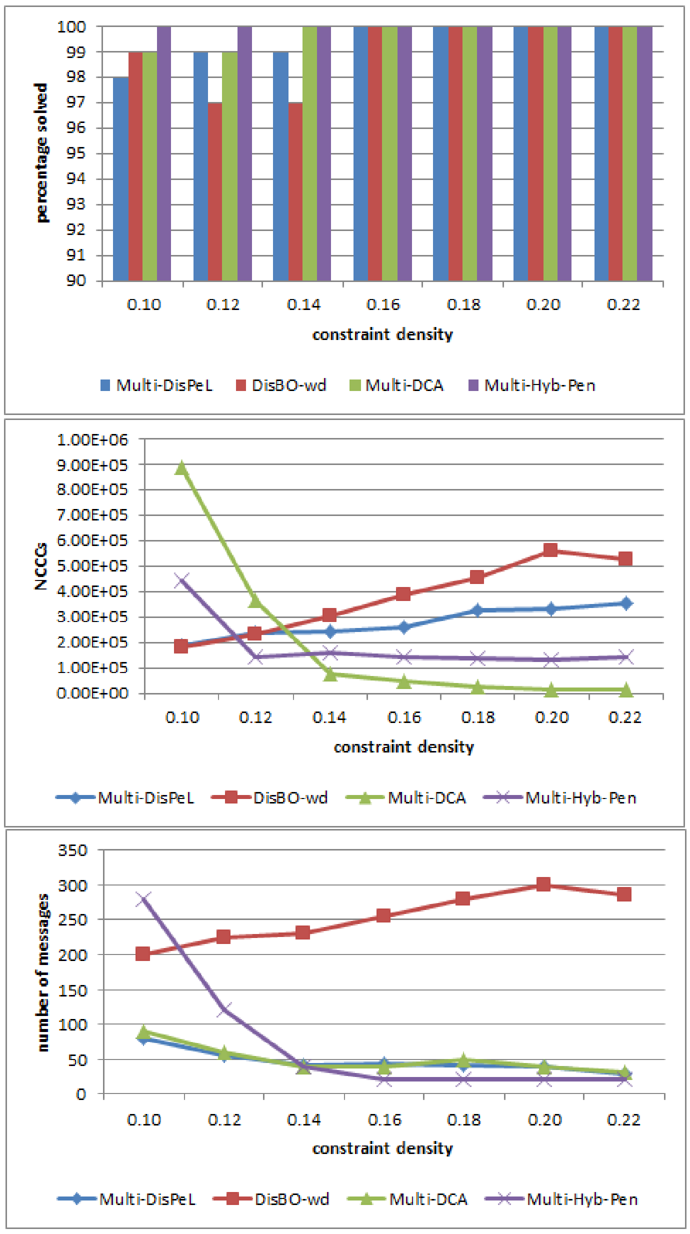


Figure 6.17: $< n = 200$, timeslots = 6, $|agents| = 10$, $d \in \{0.1, ..., 0.22\}$, md = 3 $>$

the settings, i.e. as the constraint density increases, Multi-DCA used the least number of NCCCs. However, Multi-DCA did not do well for number of messages exchanged.

- **Varying the Number of Meetings**: We use meetings $\in \{200, ..., 300\}$ with 10 agents for each problem size, a timeslot of 6, a constraint density d of 0.2 and a maximum travel time md of 3. The results are presented in Figure 6.18.

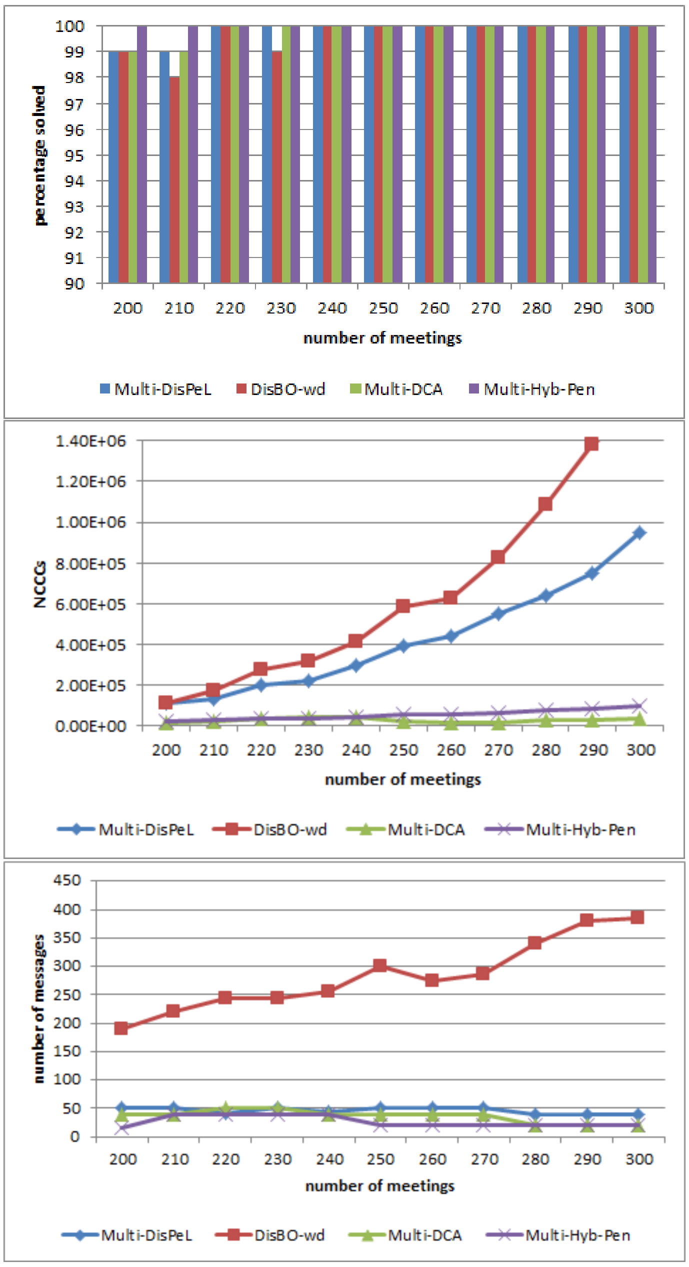


Figure 6.18: $< n \in \{200, ..., 300\}$, timeslots = 6, |agents| = 10, d = 0.2, md = 3 $>$

We fix the number of agents, thus, the number of variables per agents changed with the problem size. For example, for 200 meetings it is 20 variables per agent and so on. As the number of meetings increased, Multi-DCA had the least NCCCs compared to Multi-DisPeL and DisBO-wd and Multi-Hyb-Pen. Multi-DCA did not do well for number of messages exchanged between agents.

6.4.4 Results on Distributed Sensor Network Problems

For these experiments, we vary the number of targets $\in \{5, 6, 7, 8\}$. For each target, we use three different number of sensors n $\in \{36, 49, 64\}$ i.e. in grids of 6, 7, 8 respectively. Note: In a distributed sensor network problem, each agent is restricted to 3 variables per agent, making the CLP very simple. Thus, they are not a good representation of naturally distributed problems. However, we use distributed sensor network problems for additional evaluation of Multi-DCA. The results are presented in Table 6.5. Multi-DCA, DisBO-wd and Multi-DisPeL solved at least 97% of all the problems in each problem setting with Multi-DCA solving at least as much as DisBO-wd and Multi-Hyb-Pen. As expected, Multi-DCA used the most number of NCCCs compared to Multi-DisPeL, DisBO-wd and Multi-Hyb-Pen. However, Multi-DCA outperformed the other algorithms by having the least number of messages.

		Percentage Solved			
No. Targets	**No. Sensors**	**Multi-DisPeL**	**DisBO-wd**	**Multi-DCA**	**Multi-Hyb-Pen**
5	36	100	100	100	100
5	49	100	100	100	100
5	64	100	100	100	100
6	36	100	98	100	100
6	49	100	100	100	100
6	64	100	100	100	100
7	36	99	97	99	100
7	49	100	100	100	100
7	64	100	100	100	100
8	36	99	98	100	100
8	49	100	100	100	100
8	64	100	100	100	100
		NCCCs			
5	36	37,515	166,720	176,759	**3,527**
5	49	17,487	117,693	114,839	**3,274**
5	64	13,270	119,115	102,860	**2,888**
6	36	41,290	280,246	202,019	**4,934**
6	49	24,448	237,451	205,091	**4,934**
6	64	19,893	190,647	115,516	**3,523**
7	36	72,563	398,974	279,435	**7,412**
7	49	59,881	370,269	269,837	**4,702**
7	64	27,962	300,564	233,954	**3,712**
8	36	160,790	607,504	48 9,666	**17,659**
8	49	62,313	520,501	487,723	**7,235**
8	64	50,378	465,444	373,796	**4,743**
		Messages			
5	36	38	38	**18**	212
5	49	36	36	**12**	39
5	64	30	30	**11**	22
6	36	82	82	**26**	327
6	49	60	60	**20**	235
6	64	45	45	**16**	100
7	36	153	153	**33**	384
7	49	147	147	**28**	302
7	64	66	66	**24**	256
8	36	413	413	**48**	607
8	49	136	136	**36**	337
8	64	140	140	**32**	260

Table 6.5: $<$ sensors $\in \{36, 49, 64\}$, targets $\in \{5, 6, 7, 8\}$, $pc = 0.6$, $pv = 0.9 >$

6.5 Discussion

Divide and coordinate approach has previously been proposed for DCOPs in (Vinyals, Pujol, Rodriguez-Aguilar & Cerquides 2010), (Vinyals, Rodríguez-Aguilar & Cerquides 2010), (Hatano & Hirayama 2013) where an agent creates a copy of the problem based on their local knowledge. Each agent then solves its new sub problem and the solutions are aggregated in the coordinate stage to confirm if the solutions found by neighbouring agents are mutually the best found.

In a naturally distributed DisCSP, variables are clustered into the agents they belong to and the number of intra-agent constraints is normally higher than that of the inter-agent constraints thus, reflecting the clustering of variables in CLPs. The key idea behind the work presented in this chapter is that, when dealing with CLPs, solutions to the distributed part of the problem should be considered early in the search. An agent does not create a copy of the problem. The divide and conquer approach identifies clusters using the knowledge of the relationships (constraints) between variables. The problem is then solved with several interleaving searches. The following contributions have been made;

1. A divide and conquer approach for solving DisCSPs with complex local problems that combines several search strategies.
2. Multi-DCA - a distributed local search algorithm that implements the divide and conquer approach.
3. DynAPP-CD - a distributed local search algorithm that extends DynAPP to consider the following: (i) consider only inter-agent constraints; (ii) consider complex variables; and (iii) use compound values as domain (which are the solutions found in Stage 2 of the divide and conquer approach).
4. A formula has been derived that determines the cutoff for dividing a compound group size for distributed graph colouring problems, random DisCSP, and distributed meeting scheduling problems.

6.6 Chapter Summary

This chapter introduced a divide and conquer approach for solving DisCSPs with complex local problems that combines several search strategies: (i) systematic and local searches; (ii) centralised and distributed searches; and (iii) a modified compilation reformulation strategy. A DisCSP with CLP is divided and solved thus: (i) identifying *compound groups* in each agent; (ii) each agent finds all locally consistent solutions to each *compound group*, using one systematic search per *compound group*; (iii) using a distributed local search algorithm, the solutions found for each *compound group* are then combined with solutions to other agents' *compound groups* to form a *combination solution* such that all inter-agent constraints are satisfied; and (iv) agents check if the compound solutions to its local compound groups participating in the combination solution extends to satisfy the rest of their intra-agent constraints, to become solutions to their CLP.

We also presented Multi-DCA - Divide and Conquer Algorithm for Agents with Complex Local Problems. Multi-DCA is a distributed local search algorithm that implements the divide and conquer approach. Multi-DCA uses a basic centralised systematic search algorithm (as described in 4.2.2) in Stage 2 and Stage 4 and an adaptation of DynAPP for compound domains (DynAPP-CD) is used as the distributed local search algorithm in Stage 3. In an extensive empirical evaluation, Multi-DCA was evaluated on four problem classes and compared to state-of-the-art distributed local and systematic search DisCSP algorithms that handle CLPs. On distributed graph colouring problems, random DisCSP, and distributed meeting scheduling problems, the results of the evaluations show that the divide and conquer approach generally improved search. Multi-DCA solved at least the same number of problems compared to Multi-DisPeL and DisBO and generally incurred the least costs in the process especially with respect to time (NCCCs). As expected, on sensor network problems, Multi-DCA performed poorly on NCCCs but overall exchanged the least number of messages for all problem settings used.

Chapter 7

Conclusion and Future work

In this chapter, we give an overview of the contributions made in this thesis and we also propose some potential future research areas.

7.1 Contributions

This thesis makes contributions to the development of local search strategies for Distributed Constraint Satisfaction Problems (DisCSPs) by using combinations of existing heuristics in an improved approach to solving them. This chapter outlines our contributions and highlights possible future work.

1. **DisCSP Combination heuristics in local search for single variable/agent.**

   We proposed a novel combination heuristics which was found to be more effective in escaping local optima compared to the state of the art DisCSP algorithms. The approach combines value penalties and dynamic agent prioritisation heuristics in distributed local search where an agent represents a single variable. We presented **DynAPP** - Dynamic Agent Prioritisation with Penalties which is a synchronous distributed local search algorithm for single variable per agent problems which combines dynamic agent prioritisation with penalties on variable values (see Chapter 5). (i) Penalties are attached to individual domain values, initially set as zero. At quasi-local optima, the penalty on the ”current” values is increased, tagging the current values as bad values in order to escape from the quasi-local optima. (ii) Priorities are attached to agents, initially set to the alphabetic ordering of the agent identifier and dynamically changed, resulting in a change to the priority of the agent and, therefore, the priority of finding a consistent value for the agent's variable. Empirical results on distributed graph colouring problems, random DisCSPs and distributed meeting scheduling problems show that DynAPP generally outperformed both the competing local search algorithms (SingleDB-wd and StochDisPeL) and the complete search algorithm (PenHyb).

2. **Problem-Structure oriented problem solving for DisCSPs with CLPs**.

   We proposed a **divide and conquer approach** that improves the performance of solving DisCSPs with Complex Local Problems (CLPs) (see Chapter 6). The approach prioritises the satisfaction of the more expensive part of the problem (i.e. the inter-agent constraints). In the divide and conquer approach, the structure of the DisCSP is analysed in order to identify groups of directly-related (via an intra-agent constraint) external variables within each agent. These groups are referred to as *compound groups*. Large *compound groups* with a large number of variables and domain size are partitioned into smaller groups. A combination of systematic and local search algorithms are then run concurrently to solve the divided problem as follows: (i) each agent finds all locally consistent compound solutions to each of its *compound groups* using one centralised systematic search algorithm per *compound group*; (ii) the compound solutions found for each *compound group* are then combined with solutions to other agents' *compound groups* using a distributed local search algorithm to form a *combination solution* such that all inter-agent constraints are satisfied; and (iii) finally, using a centralised systematic search per agent, each agent checks if the compound solutions to its local compound groups (participating in the combination solution) extend to satisfy the rest of their intra-agent constraints to become solutions to its CLP. We have derived a formula to determine the optimal *compound groups* size (i.e. the number of variables in a group) and observed that a group of size 3 is generally optimal across the benchmark problems. We presented **Multi-DCA**, a distributed local search algorithm that implements the divide and conquer approach for solving DisCSPs with CLPs. We empirically evaluated Multi-DCA on distributed graph colouring problems, random DisCSPs, distributed meeting scheduling problems and distributed sensor network problems. The results show that Multi-DCA generally outperformed (by using the least time to solution) other leading DisCSPs algorithms that address complex local problems i.e. local search algorithms (Multi-DisPeL and DisBO-wd) and a hybrid complete search algorithm (Multi-PenHyb).

3. **Other contributions**

   We presented a **multi-context search** for DisCSPs where two local search algorithms (SingleDB-wd and Stoch-DisPeL) solve a given DisCSPs concurrently to improve the exploration of the search space. In the approach, each algorithm starts from a different random initialisation in a master-slave architecture. The master is responsible for finding and returning a solution or terminating if the maximum number of iterations is reached. The task of the slave is to find and store consistent partial solutions which are sent to the master solver when requested. After some specified number of iterations, the master algorithm checks with the slave algorithm if a better solution (with a lower number of constraint violations) exists and the master re-initialises its variables with the received assignments from the slave. We presented two algorithms (StochDisPeL+Restart and SingleDB-wd+Restart) that implement the multi-context search for DisCSPs having a single variable/agent. In StochDisPeL+Restart, Stoch-DisPeL is the master while SingleDB-wd is the slave; while in SingleDB-wd+Restart SingleDB-wd is the master while StochDisPeL is the slave. Based on an empirical evaluation on several problem instances, we found that StochDisPeL+Restart generally performed better than StochDisPeL and close to DynAPP, although less prominent than DynAPP. On the other hand, SingleDB-wd+Restart performed better than only SingleDB-wd.

7.2 Future Work

In this section, we point out some possible future direction for this work.

7.2.1 Extending the approach of handling DisCSPs with CLPs

Although the divide and conquer approach presented in this thesis was found to be beneficial, alternative search methods and adaptive problem solving could be investigated.

Using alternative search methods

In our implementation of the divide and conquer approach (Multi-DCA), we used an adaptation of DynAPP as the distributed local search algorithm. Other distributed local search algorithms such as DisBO-wd could be used. This will require DisBO-wd to be modified for solving DisCSPs with compound variables having a domain of compound values.

Adaptive problem solving

The problem structure of a DisCSP with CLPs was exploited in a novel divide and conquer approach. The structure could be further exploited for addressing each complex local problem using a different approach, such as selecting an approach based on the number of external variables and constraints.

7.2.2 Alternative Approach in the Multi-Context Search

The Multi-Context search algorithms presented performed better in terms of both communication cost and computational effort when compared to the algorithms without restart. Alternative approaches for restart could be investigated such as: (i) the master algorithm restarting when there are no improvements; or (ii) the master algorithm restarting the values for only certain parts of the problem. Non-improvement could be determined by the master solver using techniques such as the number of local optima detected.

7.2.3 DynAPP and Multi-DCA for Optimisation

DynAPP and Multi-DCA are algorithms for DisCSPs where the first solution that satisfies all constraints is returned. In a DCOPs, a solution minimises or maximises a given objective function. It would be interesting to determine the performance of DynAPP and Multi-DCA on optimization problems. Challenges of using our algorithms to solve DCOPs include the guarantee on solution quality. For example, in MV-MC-DCOPs, each agent controls two or more variables and an integer constant is used to represent the minimum number of satisfied internal variables for the assignment returned by the given agent to be considered useful overall. The divide and conquer approach returns quick solutions to the distributed problem and should scale well for MV-MC-DCOP.

7.2.4 Dynamic real-time DisCSPs

Dynamic DisCSPs address situations where the problem definition changes during search. New variables, domain values or constraint could be added or dropped during search. These changes can be captured as a series of constraint modifications (Verfaillie & Jussien 2005). This imposes challenges on finding an efficient approach to solving such dynamic modifications without abandoning existing solutions found from previous searches. Currently, in our divide and conquer approach for DisCSPs with complex local problems, depending on where the change in the problem specification occurs, the algorithm may reuse or restart from scratch. It would be interesting to investigate the performance of our divide and conquer approach on Dynamic DisCSPs.

7.3 Summary

This thesis has investigated heuristics and strategies in distributed local search. Several novel local search algorithms and combination heuristics for distributed constraint satisfaction have been presented. The main aim of this thesis is to investigate if there are any benefits in combining existing strategies for distributed local search (as stated in Chapter 1, Section 1.1). The research objectives are:

- Investigate the effect of combining existing search heuristics to escape local optima in local search algorithms for distributed constraint satisfaction problems.
- Exploit the structure of distributed constraint satisfaction problems with complex local problems to develop a more efficient approach to solving them.

We have investigated several combination strategies for solving DisCSPs and present **DynAPP** and **Multi-DCA** which meet these objectives as follows;

- We have presented a novel algorithm - **DynAPP** that uses a combination of heuristics to escape local optima for the resolution of distributed constraint satisfaction problems where each agent represents a single variable.
- We have also exploited the structure of distributed constraint satisfaction problems with CLPs and proposed a **divide and conquer approach** to solving them that combines several search strategies. **Multi-DCA** is a successful algorithm with an overall distributed local search algorithm that implements the approach.

Both algorithms were found to generally reduce the time (in the form of non-concurrent computational steps or NCCCs) and communication load (in the form of the number of messages exchanged) in solving distributed constraint satisfaction problems. In conclusion, *"We have investigated several existing combination heuristics and strategies and have shown that there are indeed benefits in combining search strategies and heuristics in local search for solving DisCSPs"*.

Bibliography

Aji, S. & McEliece, R. (2000). The generalized distributive law, *Information Theory, IEEE Transactions on* **46**(2): 325–343.

Armstrong, A. & Durfee, E. (1997). Dynamic prioritization of complex agents in distributed constraint satisfaction problems, *IN PROCEEDINGS OF 15TH IJCAI*, pp. 620–625.

Bacchus, F. & Run, P. v. (1995). Dynamic variable ordering in csps, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, CP '95, Springer-Verlag, London, UK, UK, pp. 258–275.

Basharu, M., Arana, I. & Ahriz, H. (2005). Solving discsps with penalty driven search, *Proceedings of the 20th national conference on Artificial intelligence - Volume 1*, AAAI Press, pp. 47–52.

Basharu, M., Arana, I. & Ahriz, H. (2006). Stochdispel: exploiting randomisation in dispel, *Proceedings of 7th International Workshop on Distributed Constraint Reasoning, DCR2006*, pp. 117–131.

Basharu, M., Arana, I. & Ahriz, H. (2007a). Escaping local optima: constraint weights vs. value penalties, *27th SGAI International Conference on Artificial Intelligence, AI-07*, Springer, pp. 51–64.

Basharu, M., Arana, I. & Ahriz, H. (2007b). Solving coarse-grained discsps with multi-dispel and disbo-wd, *IEEE / WIC / ACM International Conference on Intelligent Agent Technology* **0**: 335–341.

Bessière, C. (1999). Non-binary constraints, *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, pp. 24–27.

Bessière, C., Brito, I., Maestre, A. & Meseguer, P. (2005). Asynchronous backtracking without adding links: A new member in the abt family, *Artificial Intelligence* **161**(1-2): 7–24.

Bowring, E. & Tambe, M. (2006). Multiply-constrained distributed constraint optimization, *In AAMAS*, pp. 1413–1420.

Brélaz, D. (1979). New methods to color the vertices of a graph, *Commun. ACM* **22**(4): 251–256.

Burke, D. (2008). *Exploiting Problem Structure in Distributed Constraint Optimization with Complex Local Problems*, PhD thesis, National University of Ireland, Cork.

Burke, D. A. & Brown, K. N. (2006a). A comparison of approaches to handling complex local problems in dcop, *In Distributed Constraint Satisfaction Workshop, Riva del Garda, Italy*, p. 27?33.

Burke, D. A. & Brown, K. N. (2006b). Efficiently handling complex local problems in distributed constraint optimisation, pp. 701–702.

Burke, D. A. & Brown, K. N. (2007). Using relaxations to improve search in distributed constraint optimisation, *Artificial Intelligence Review* **28**(1): 35–50.

Davin, J. & Modi, P. J. (2006). Hierarchical variable ordering for distributed constraint optimization, *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, ACM, New York, NY, USA, pp. 1433–1435.

Dechter, R. (2003). *Constraint Processing*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Duong, T.-T., Pham, D. N., Sattar, A. & Newton, M. A. H. (2013). Weight-enhanced diversification in stochastic local search for satisfiability, *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, AAAI Press, pp. 524–530.

Eisenberg, C. (2003). *Distributed Constraint Satisfaction For Coordinating And Integrating A Large-Scale, Heterogeneous Enterprise*, PhD thesis, Swiss Federal Institute of Technology (Ecole Polytechnique Federale De Lusanne).

Faltings, B., Leaute, T. & Petcu, A. (2008). Privacy guarantees through distributed constraint satisfaction, *Web Intelligence and Intelligent Agent Technology,WI-IAT*, Vol. 2, pp. 350–358.

Farinelli, A., Rogers, A., Petcu, A. & Jennings, N. R. (2008). Decentralised coordination of low-power embedded devices using the max-sum algorithm, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 639–646.

Fitzpatrick, S. & Meertens, L. G. L. T. (2001). An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs, *Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications*, SAGA '01, Springer-Verlag, London, UK, UK, pp. 49–64.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search, *J. ACM* **29**(1): 24–32.

Freuder, E. C. & Wallace, R. J. (1992). Partial constraint satisfaction, *Artificial Intelligence* **58(1–3)**: 21–70.

Gershman, A., Meisels, A. & Zivan, R. (2006). Asynchronous forward-bounding for distributed constraints optimization, *In: Proc. 1st Intern. Workshop on Distributed and Speculative Constraint Processing. (2005*.

Glover, F. (1990). Tabu search – part ii, *ORSA J. on Computing* **2**: 4–32.

Grinshpoun, T., Grubshtein, A., Zivan, R., Netzer, A. & Meisels, A. (2013). Asymmetric distributed constraint optimization problems, *J. Artif. Intell. Res. (JAIR)* **47**: 613–647.

Hamadi, Y. (1998). Backtracking in distributed constraint networks, *International Journal on Artificial Intelligence Tools*, pp. 219–223.

Hamadi, Y. (2002). Interleaved backtracking in distributed constraint networks.

Haralick, R. M. & Elliott, G. L. (1979). Increasing tree search efficiency for constraint satisfaction problems, *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 356–364.

Hatano, D. & Hirayama, K. (2013). Deqed: An efficient divide-and-coordinate algorithm for dcop, *in* F. Rossi (ed.), *IJCAI*, IJCAI/AAAI.

Hirayama, K. & Toyoda, J. (1995). Forming Coalitions for Breaking Deadlocks, *International Conference on Multiagent Systems*, pp. 155–162.

Hirayama, K. & Yokoo, M. (1997). Distributed partial constraint satisfaction problem, *Principles and Practice of Constraint Programming*, pp. 222–236.

Hirayama, K. & Yokoo, M. (2000). An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction, *In Proceedings of International Conference on Multiagent Systems*, pp. 135–142.

Hirayama, K. & Yokoo, M. (2002). Local search for distributed sat with complex local problems, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3*, AAMAS '02, ACM, New York, NY, USA, pp. 1199–1206.

Hirayama, K. & Yokoo, M. (2005). The distributed breakout algorithms, *Artif. Intell.* **161**: 89–115.

Hirayama, K., Yokoo, M. & Sycaraw, K. (2004). An easy-hard-easy cost profile in distributed constraint satisfaction(knowledge processing), *IPSJ Journal* **45**(9): 2217–2225.

Ismel, B. (2007). *Distributed Constraint Satisfaction*, PhD thesis, Institut d'Investigacio en Intel.ligencia Artificial Consejo Superior de Investigaciones Cientificas.

Khanna, S., Sattar, A., Hansen, D. & Stantic, B. (2009). An efficient algorithm for solving dynamic complex dcop problems, *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT '09. IEEE/WIC/ACM International Joint Conferences on*, Vol. 2, pp. 339–346.

Kiekintveld, C., Yin, Z., Kumar, A. & Tambe, M. (2010). Asynchronous algorithms for approximate distributed constraint optimization with quality bounds, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 133–140.

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

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system, *Commun. ACM* **21**(7): 558–565.

Lee, D. A. J. (2010). *Hybrid algorithms for distributed constraint satisfaction*, PhD thesis, School of Computing, Robert Gordon University, Aberdeen.

Lee, D., Arana, I., Ahriz, H. & Hui, K. (2009a). A hybrid approach to solving coarse-grained discsps, *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1235–1236.

Lee, D., Arana, I., Ahriz, H. & Hui, K.-Y. (2009b). Multi-hyb: A hybrid algorithm for solving discsps with complex local problems, *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on*

Web Intelligence and Intelligent Agent Technology - Volume 02, WI-IAT '09, IEEE Computer Society, Washington, DC, USA, pp. 379–382.

Maestre, A. & Bessière, C. (2004). Improving asynchronous backtracking for dealing with complex local problems.

Maheswaran, R. T., Pearce, J. P. & Tambe, M. (2004). Distributed algorithms for dcop: A graphical-game-based approach, *In PDCS*.

Mailler, R. & Lesser, V. (2003). A mediation based protocol for distributed constraint satisfaction, *In Workshop on DCR*.

Mailler, R. & Lesser, V. (2004). Solving distributed constraint optimization problems using cooperative mediation, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, IEEE Computer Society, Washington, DC, USA, pp. 438–445.

Mailler, R. & Lesser, V. R. (2006). Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems, *Journal of Artificial Intelligence Research (JAIR* **2005**: 2006.

Meisels, A., Kaplansky, E., Razgon, I. & Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms, *In: Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pp. 86–93.

Minton, S., Johnston, M. D., Philips, A. B. & Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence* **58**: 161 – 205.

Modi, P. J., Shen, W.-M., Tambe, M. & Yokoo, M. (2005). Adopt: asynchronous distributed constraint optimization with quality guarantees, *Artificial Intelligence* **161**: 149 – 180. Distributed Constraint Satisfaction.

Morris, P. (1993). The breakout method for escaping from local minima, *Proceedings of the eleventh national conference on Artificial intelligence*, AAAI'93, AAAI Press, pp. 40–45.

Mueller, R. & Havens, W. S. (2005). Queuing local solutions in distributed constraint satisfaction systems, *Advances in Artificial Intelligence*, Vol. 3501 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 103–107.

Palmer, E. M. (1985). *Graphical Evolution: An Introduction to the Theory of Random Graphs*, John Wiley and Sons, Inc.

Pearce, J. P. & Tambe, M. (2007). Quality guarantees on k-optimal solutions for distributed constraint optimization problems, *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 1446–1451.

Petcu, A. & Faltings, B. (2005). A scalable method for multiagent constraint optimization, *In Proc. of IJCAI*, pp. 1413–1420.

Portway, C. & Durfee, E. H. (2010). The multi variable multi constrained distributed constraint optimization framework, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1385–1386.

Rogers, A., Farinelli, A., Stranders, R. & Jennings, N. (2011). Bounded approximate decentralised coordination via the max-sum algorithm, *Artificial Intelligence* **175**(2): 730 – 759.

Rollon, E. & Larrosa, J. (2012). Improved bounded max-sum for distributed constraint optimization, *in* M. Milano (ed.), *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 624–632.

Rossi, F., Beek, P. v. & Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science Inc., New York, NY, USA.

Schiex, T., Fargier, H. & Verfaillie, G. (1995). Valued constraint satisfaction problems: Hard and easy problems, *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 631–637.

Silaghi, M. C. & Yokoo, M. (2006). Adopt-ng: Unifying asynchronous distributed optimization with asynchronous backtracking.

Smith, M. & Mailler, R. (2010). Improving the efficiency of the distributed stochastic algorithm, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1417–1418.

Steven, M., Mark, D. J., Andrew, B. P. & Philip, L. (1992). Minimizing conflicts:a heuristic repair method for constraint-satisfaction and scheduling problems., *Artificial Intelligence* p. 58(1173):16117205.

Verfaillie, G. & Jussien, N. (2005). Constraint solving in uncertain and dynamic environments: A survey, *Constraints* **10**(3): 253–281.

Vinyals, M., Pujol, M., Rodriguez-Aguilar, J. A. & Cerquides, J. (2010). Divide-and-coordinate: Dcops by agreement, *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 149–156.

Vinyals, M., Rodríguez-Aguilar, J. A. & Cerquides, J. (2010). Divide-and-coordinate by egalitarian utilities: turning dcops into egalitarian worlds, Toronto, Canada. May 2010.

Vinyals, M., Shieh, E., Cerquides, J., Rodriguez-Aguilar, J. A., Yin, Z., Tambe, M., & Bowring, E. (2011). Quality guarantees for region optimal dcop algorithms, *International Conference on Autonomous Agents and Multiagent Systems*.

Yeoh, W., Felner, A. & Koenig, S. (2014). Bnb-adopt: An asynchronous branch-and-bound DCOP algorithm, *CoRR* **abs/1401.3490**.

Yokoo, M. (1995a). Asynchronous weak-commitment search for solving distributed constraint satisfaction problems, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag, London, UK, pp. 88–102.

Yokoo, M. (1995b). Asynchronous weak commitment search for solving large-scale distributed constraint satisfaction problems, *Proceedings of the first international conference on multiagent systems*, AAAI'95, AAAI Press.

Yokoo, M., Durfee, E. H., Ishida, T. & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms, *IEEE Trans. on Knowl. and Data Eng.* **10**: 673–685.

Yokoo, M. & Hirayama, K. (1996). Distributed breakout algorithm for solving distributed constraint satisfaction problems.

Yokoo, M. & Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review, *Autonomous Agents and Multi-Agent Systems* **3**(2): 185–207.

Zhang, W., Wang, G. & Wittenburg, L. . (2002). Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance, *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*.

Zhang, W., Wang, G., Xing, Z. & Wittenburg, L. (2005). Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks, *Artificial Intelligence* **161**(1,2): 55 – 87. Distributed Constraint Satisfaction.

Zhang, W. & Wittenburg, L. (2002). Distributed breakout revisited, *Eighteenth national conference on Artificial intelligence*, American Association for Artificial Intelligence, Menlo Park, CA, USA, pp. 352–357.

Zhou, L., Thornton, J. & Sattar, A. (2003). Dynamic agent ordering in distributed constraint satisfaction problems., *Australian Conference on Artificial Intelligence*, Vol. 2903 of *Lecture Notes in Computer Science*, Springer, pp. 427–439.

Zivan, R. (2008). Anytime local search for distributed constraint optimization, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1449–1452.

Zivan, R. & Meisels, A. (2003). Synchronous vs asynchronous search on discsps, *Proceedings of the First European Workshop on Multi-Agent Systems (EUMAS)*.

Appendix A

Combination Heuristics Algorithms

A.1 Introduction

In this Appendix, we discuss the combination of constraint weights, value penalty and dynamic agent prioritisation in local search algorithms for DisCSPs, where an agent represents a single variable. We describe the algorithms implemented with this combination heuristics and present results of an empirical evaluation on several problem classes. As part of the combination of heuristics in local search algorithm, the multi-context search with constraint weights and value penalties is presented in Appendix B. A complete discussion on the algorithm that combines dynamic agent prioritisation and value penalties strategy (DynAPP) was presented in Chapter 5 as this is the algorithm which, overall, gave best results. Next we briefly describe the heuristics used in Section A.2, the implementations are discussed in Section A.3 and the empirical evaluation in Section A.4. We summarise the appendix in Section A.5.

A.2 Heuristics Considered

In this section, we describe the heuristics we use in our combination strategies. When used individually, constraint weights, value penalty and dynamic agent priority have been found to be effective in escaping local optima. Our hypothesis is that in combining these heuristics we would further improve how local optima is handled thus improve the efficiency of distributed local search algorithms.

Constraint weights is a breakout mechanism used to escape quasi-local optima by increasing the weights attached to constraints that are violated at a quasi-local optima, hence, making the satisfaction of these constraints more important. This strategy was implemented in SingleDB (Hirayama & Yokoo 2005), a distributed local search algorithm for single variable per agent that escapes quasi-local optima with a weight decay mechanism that decreases constraint weights at every iteration, thus "forgetting" weights over time. Refer to Chapter 4, Section 4.3.1 for a discussions on SingleDB-wd.

Penalties on value imposes penalties on the assignments of variables found to be inconsistent at quasi-local optima. Thus, the heuristic identifies and avoids bad values that led to the local optima. Stoch-DisPeL (Basharu et al. 2006) is a distributed local search algorithms for single variable per agent that imposes penalties on values to escape quasi-local optima. Refer to Chapter 4, Section 4.3.1 for a discussions on Stoch-DisPeL.

Dynamic agent prioritisation heuristic on the other hand increases the importance of an agent over its neighbours when the agents variables are involved in local optima. Asynchronous Weak Commitment Search (AWCS)(Yokoo 1995a) is a complete distributed algorithm for single variable per agent that implements this heuristic. Refer to Chapter 4, Section 4.3.3 for a discussions on AWCS.

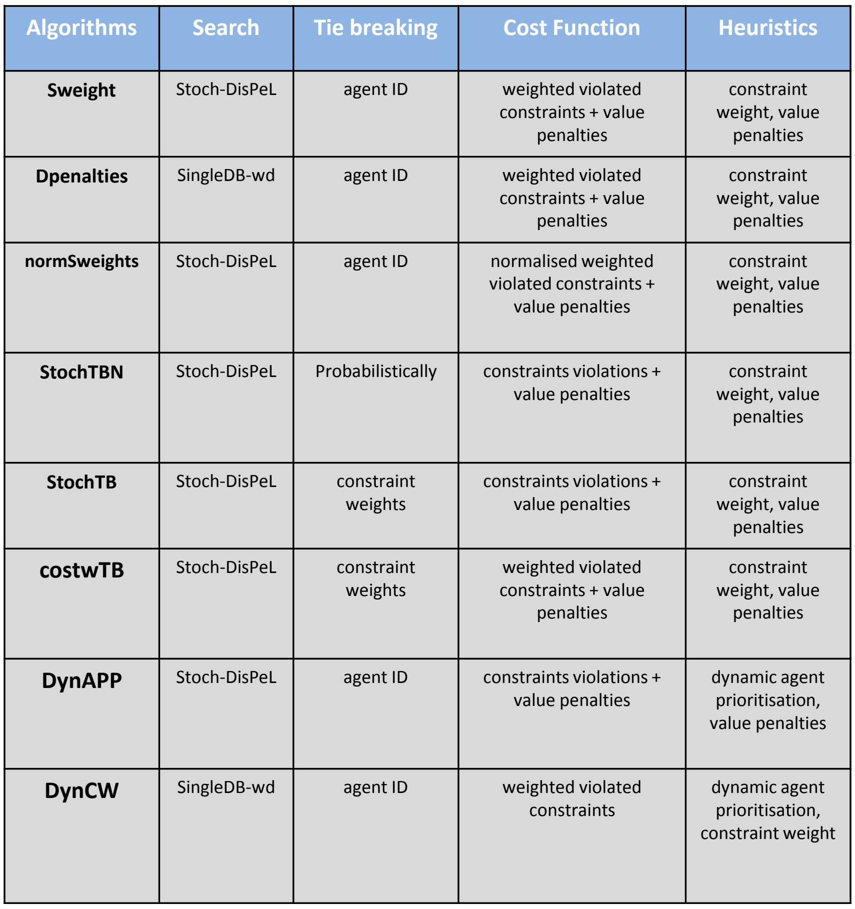
A.3 Implementations

We propose a number of combinations heuristics with constraint weights, dynamic agent prioritisation and value penalties to implement distributed local search algorithms for DisCSPs having single variable per agent. The algorithms with the combination heuristics differ on what heuristic is used to (i) compute the cost function; and/or (ii) to break ties. We also investigated the effect of normalisation in some of the heuristics. The algorithms we implement with the combination heuristics use the search technique in either Stoch-DisPeL or SingleDB-wd. Stoch-DisPeL (Basharu et al. 2006) and SingleDB-wd (Lee 2010) are distributed local search algorithms for single variable per agent that impose penalties on values and constraint weights respectively to escape quasi-local optima. Refer to Chapter 4, Section 4.2.3 for a further discussion on SingleDB-wd and Stoch-DisPeL. We present a summary of the algorithms with the combination heuristics in Figure A.1 and describe them next.

1. Constraint Weights and Value Penalty to Cost Function: Sweight, Dpenalties and normSweights

   In these algorithms, the combination heuristics are used to evaluate the cost of selecting a value.

   (a) **(Sweight)** - Stochastic Distributed Penalty Driven Search with Weight: The search algorithm is similar to Stoch-DisPeL and the weights of constraints violated at quasi-local optima are



Algorithms	Search	Tie breaking	Cost Function	Heuristics
Sweight	Stoch-DisPeL	agent ID	weighted violated constraints + value penalties	constraint weight, value penalties
Dpenalties	SingleDB-wd	agent ID	weighted violated constraints + value penalties	constraint weight, value penalties
normSweights	Stoch-DisPeL	agent ID	normalised weighted violated constraints + value penalties	constraint weight, value penalties
StochTBN	Stoch-DisPeL	Probabilistically	constraints violations + value penalties	constraint weight, value penalties
StochTB	Stoch-DisPeL	constraint weights	constraints violations + value penalties	constraint weight, value penalties
costwTB	Stoch-DisPeL	constraint weights	weighted violated constraints + value penalties	constraint weight, value penalties
DynAPP	Stoch-DisPeL	agent ID	constraints violations + value penalties	dynamic agent prioritisation, value penalties
DynCW	SingleDB-wd	agent ID	weighted violated constraints	dynamic agent prioritisation, constraint weight

Figure A.1: Algorithms with combination heuristics

increased by 1.

(b) **(Dpenalties)**: Single Distributed Breakout with Weight Decay and Penalties: The search algorithm is similar to SingleDB-wd and additionally, temporary penalties (a fixed value 3) are also imposed on the values of the variable's found to be inconsistent at quasi-local optima. Temporary penalties are reset when an agent selects a new value.

(c) **(normSweights)**: Normalised Stochastic Distributed Penalty Driven Search Weights: This is a variation of Sweight. The difference is weights are normalised (to remove scale bias) before adding them to the cost function.

Sweight, Dpenalties and normSweights use the default strategy of their search algorithm by selecting the agent with the lowest ID for *breaking ties*.

Cost function The cost of selecting a given domain value is calculated as the weighted number of constraint violations and penalties on values (see Equation A.1).

$$f(d_j) = \sum_{i=1}^{k}(cw_i * viol(c_i)) + p(d_j) \qquad {}_i \in \{1, ..., k\} \tag{A.1}$$

where c_i is the ${}_i$th constraint

d_j is the ${}_j$th value in the variables domain

cw_i is the constraint weight of the ${}_i$th constraint

$viol(c_i)$ is the number of constraints violated

$p(d_j)$ is the penalty attached to d_j

2. Constraint Weights and Value Penalty for Tie Breaking: StochTB, StochTBP and costwTB

   In these combinations, the algorithms search for a solution similar to Stoch-DisPeL with penalties on values for escaping quasi-local optima. Constraint weights are also implemented at quasi-local optima, increasing them by 1.

   Tie Breaking is done as follows

   (a) **(StochTB)**: Stochastic Distributed Penalty Driven Search with Tie Breaking: Ties are broken with the constraint weights. To break a tie between 2 values (v_1,v_2), the total weights on the constraints violated by each value is calculated and the value with the least weighted constraint violations is selected.

   (b) **(StochTBP)**: Stochastic Distributed Penalty Driven Search with Probabilistic Tie Breaking: Ties are broken based on some probability (p). Based on an empirical evaluation with several values, we found p = 0.3 generally performed best. To break a tie between 2 values (v_1,v_2), a value v_1 is selected with probability p or the value v_2 is retained otherwise.

   Cost function for StochTBP and StochTB is determined by the number of constraint violations and penalties on values (see Equation A.2). Constraints are not weighted.

$$f(d_j) = viol(d_j) + p(d_j) \qquad j \in \{1..|domain|\} \tag{A.2}$$

   where d_j is the ${}_j$th value in the domain

   $viol(d_j)$ is the number of constraints violated

$p(d_j)$ is the penalty imposed on d_j

(c) **(costwTB)**: Stochastic Distributed Penalty Driven Search with Weights for tie breaking cost: costwTB uses a combination of (a) tie breaking with weights (see StochTB above); and (b) normalized weights added to the cost function (see normSweights above).

3. **(DynAPP)**: Dynamic Agent Prioritisation with Penalties: The search algorithm is similar to Stoch-DisPeL. In DynAPP, the two heuristics combined are dynamic agent prioritisation and value penalty as follows: (i) A priority is attached to each agent; and (ii) penalties are attached to individual domain values. At quasi-local optima, the penalty on the current assignment of variables with constraint violations is increased and the priority of the agent is also changed. Ties are broken by selecting the agent with the lowest ID. The *cost function* is determined by the number of constraint violations and penalties on values (see Equation A.3).

$$f(d_j) = viol(d_j) + p(d_j) \qquad j \in \{1..|domain|\} \tag{A.3}$$

where d_j is the $_j$th value in the domain

$viol(d_j)$ is the number of constraints violated

$p(d_j)$ is the penalty imposed on d_j

4. **(DynCW)**: Dynamic Agent Prioritisation with Constraint Weights: The search algorithm is similar to SingleDB-wd. In DynCW, the two heuristics combined are dynamic agent prioritisation and constraint weights as follows: (i) A priority is attached to each agent; and (ii) weights are attached to each constraint. At quasi-local optima, the weights on violated constraints are increased and the priority of the agent is also changed. Ties are broken by selecting the agent with the lowest ID. The *cost function* is calculated as the weighted number of constraint violations (see Equation A.4).

$$f(d_j) = \sum_{i=1}^{k}(cw_i * viol(c_i)) \qquad {}_i \in \{1, ..., k\} \tag{A.4}$$

where c_i is the $_i$th constraint

d_j is the $_j$th value in the variables domain

cw_i is the constraint weight of the $_i$th constraint

$viol(c_i)$ is the number of constraints violated if d_j is selected

A.4 Empirical Evaluation of Combination Heuristics Algorithms

We empirically evaluate Sweight, Dpenalties, normSweights, StochTBP, StochTB and costwTB, DynAPP and DynCW on **distributed graph colouring problems**, **random DisCSPs** and **distributed meeting scheduling problems** (refer to Chapter 3 for the description of the problems). Because the algorithms implemented are for solving DisCSP where an agent represents a single variable, we exempt sensor network problems from our evaluation due to its inherent nature of having 3 variables per agent. For each problem setting, we run 100 solvable problem instances from which we record and present (i) the percentage of problems solved within the maximum number of iterations; (ii) the median number of Non-Concurrent Constraint Checks (NCCCs) performed; and (iii) the median number of messages sent. Although CPU time is not an established measure for DisCSPs (Meisels et al. 2002) and is not reported in this thesis, we measured CPU time and found the results to be consistent with the other evaluation metrics used. Dpenalties and DynCW solved less than 50% for most problem settings while using an exponential number of messages and NCCCs. To highlight the performance of algorithms that solved more problems, Dpenalties and DynCW were not included in our illustrations of the results as it would obscure the differences in performance for the other algorithms.

The results are benchmarked with two other local search algorithms that have a strategy for escaping local optima; **Stoch-DisPeL** and **SingleDB-wd**. We excluded DSA (Zhang et al. 2002) due to the fact that it has no explicit mechanism for escaping from local optima, thus, does not perform well where the goal is to satisfy all constraints. Once stuck at local optima, the sideways moves are usually insufficient to push a search out of locally optimal regions. Each algorithm was allowed to run for a maximum of (100 * (n)) iterations where n represents the number of variables (nodes, variables or meetings). Note: In the few cases where not all problems were solved, the effort "wasted" in that problem are not counted towards the median number of messages and median NCCCs, they do however affect the percentage of problems solved.

Results on Distributed Graph Colouring Problems

We used distributed graph colouring problems with 3 colours, 200 nodes and we vary the degree deg $\in \{4.3, ..., 5.3\}$ in steps of 0.1 to determine the performance on problems with varying difficulty. The results exhibited the easy-hard-easy trend with less problems being solved at the phase transition (where the problems are more difficult) i.e at deg ∈ {4.7, 4.8, 4.9} (see Figure A.2). DynAPP solved at least as many problems as the other algorithms in all cases. For both number of messages and NCCCs, StochTB, Sweight and DynAPP performed better than Stoch-DisPeL with DynAPP as the overall best.

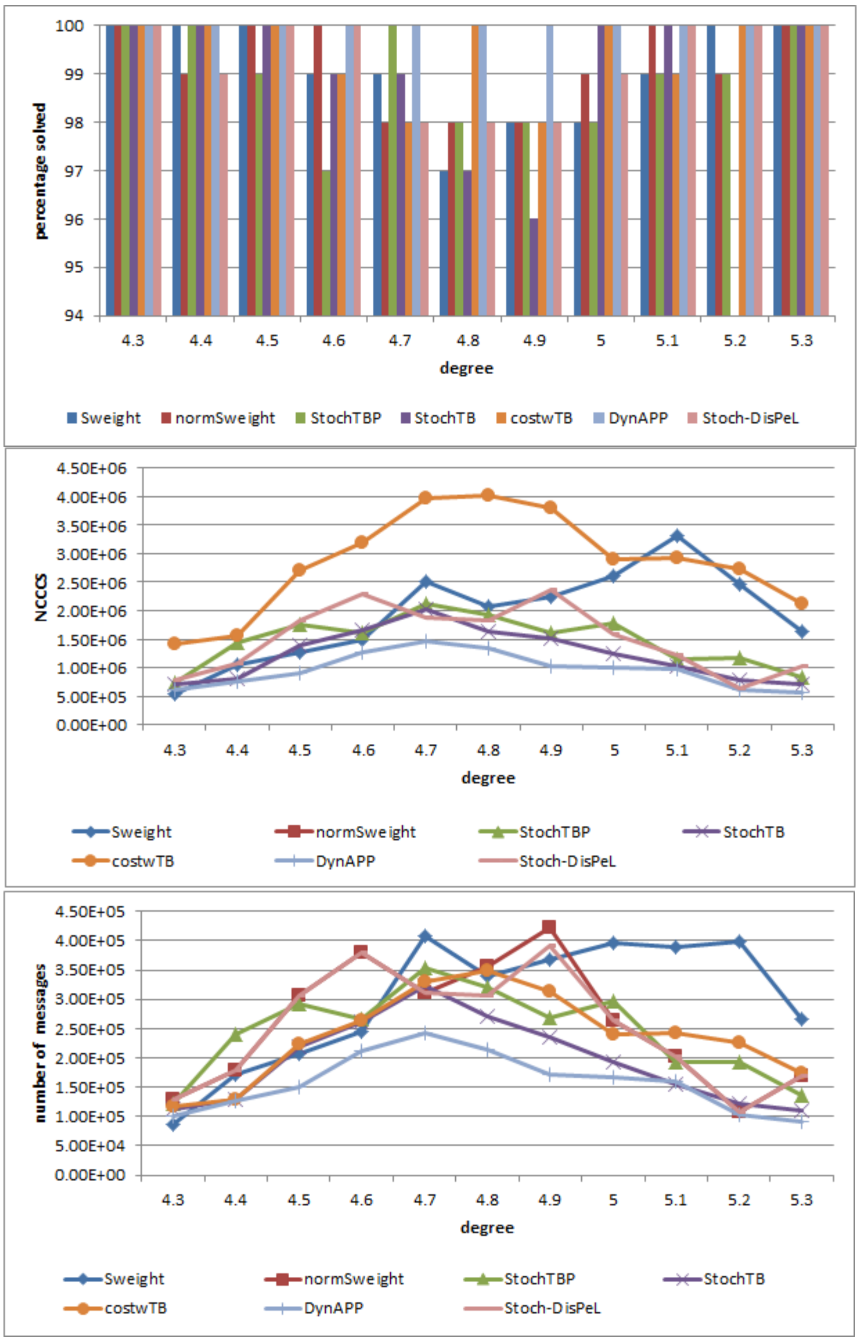


Figure A.2: $< n = 200, |\text{colours}| = 3, \text{deg} \in \{4.3, ..., 5.3\} >$

Results on Random Distributed Constraint Satisfaction Problems

For random DisCSPs, we use 200 variables, domain size of 8, a constraint density p_1 of 0.15 and we vary constraint tightness $p_2 \in \{0.1, ..., 0.9\}$ in steps of 0.1. We vary constraint tightness to determine the performance on random DisCSPs with varying difficulty. As illustrated in Figure A.3, we observe that at the phase transition i.e. around p_2 of 0.4, all the algorithms solved the least number of problems and used the most number of messages and NCCCs. However, StochTB, Sweight and DynAPP performed better than Stoch-DisPeL both on NCCCs and number of messages exchanged. Overall, DynAPP generally performed best.

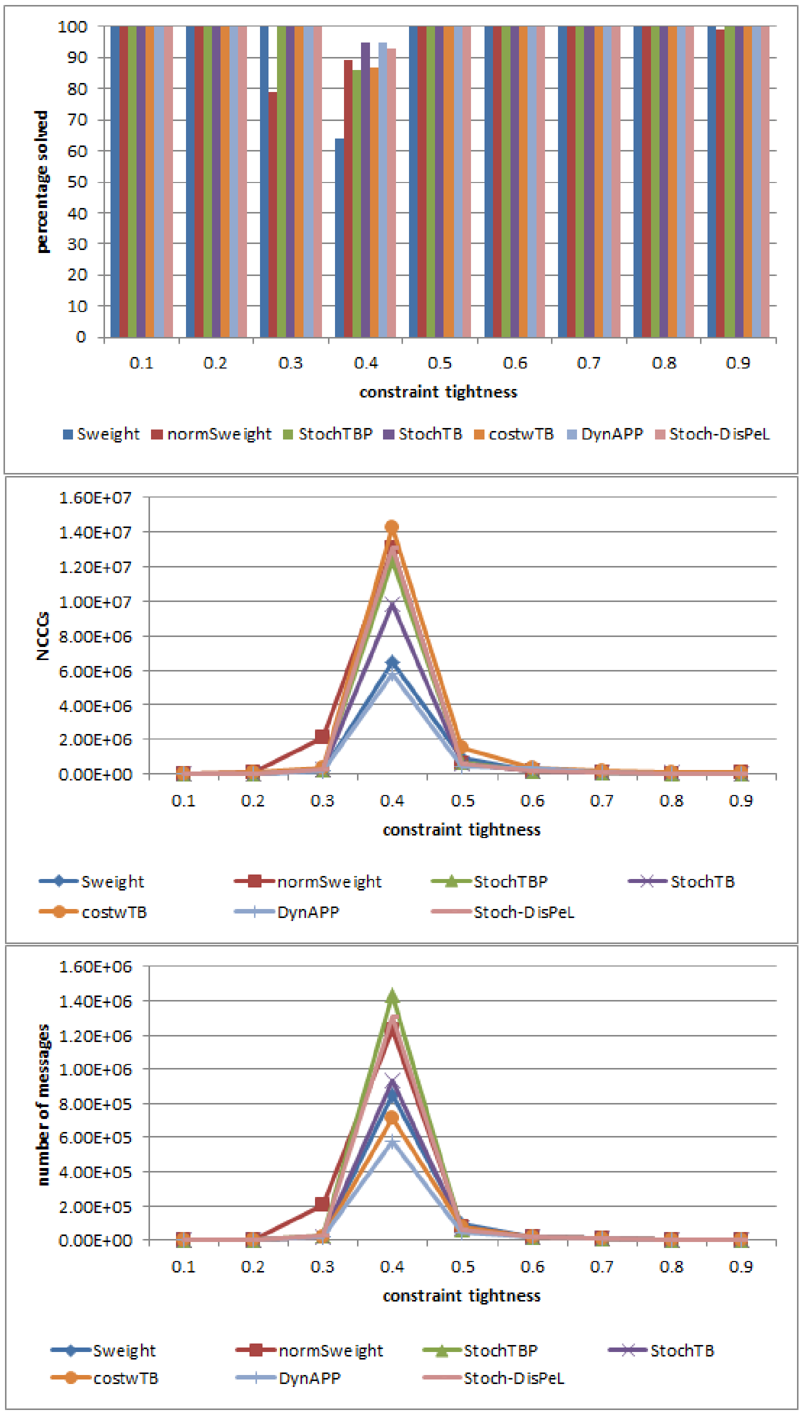


Figure A.3: $< n = 200, |\text{domain}| = 8, p_1 = 0.15, p_2 \in \{0.1, .., 0.9\} >$

Results on Distributed Meeting Scheduling Problems

For distributed meeting scheduling problems, we used 200 meetings, a density $p_1 \in \{0.1, ..., 0.2\}$ in steps of 0.01 and a maximum possible distance md of 3. We vary constraint density to determine the performance on problems with varying difficulty. The results are presented in Figure A.4. Similar to the earlier results, some of the combination heuristics exhibited improvements compared to Stoch-DisPeL. Although most of the problems were solved, Sweight, normSweight, StochTBP, StochTB and DynAPP had similar number of messages and NCCCs. It is observed that, around p_1 = 0.14, 0.15, DynAPP had the least messages and NCCCs.

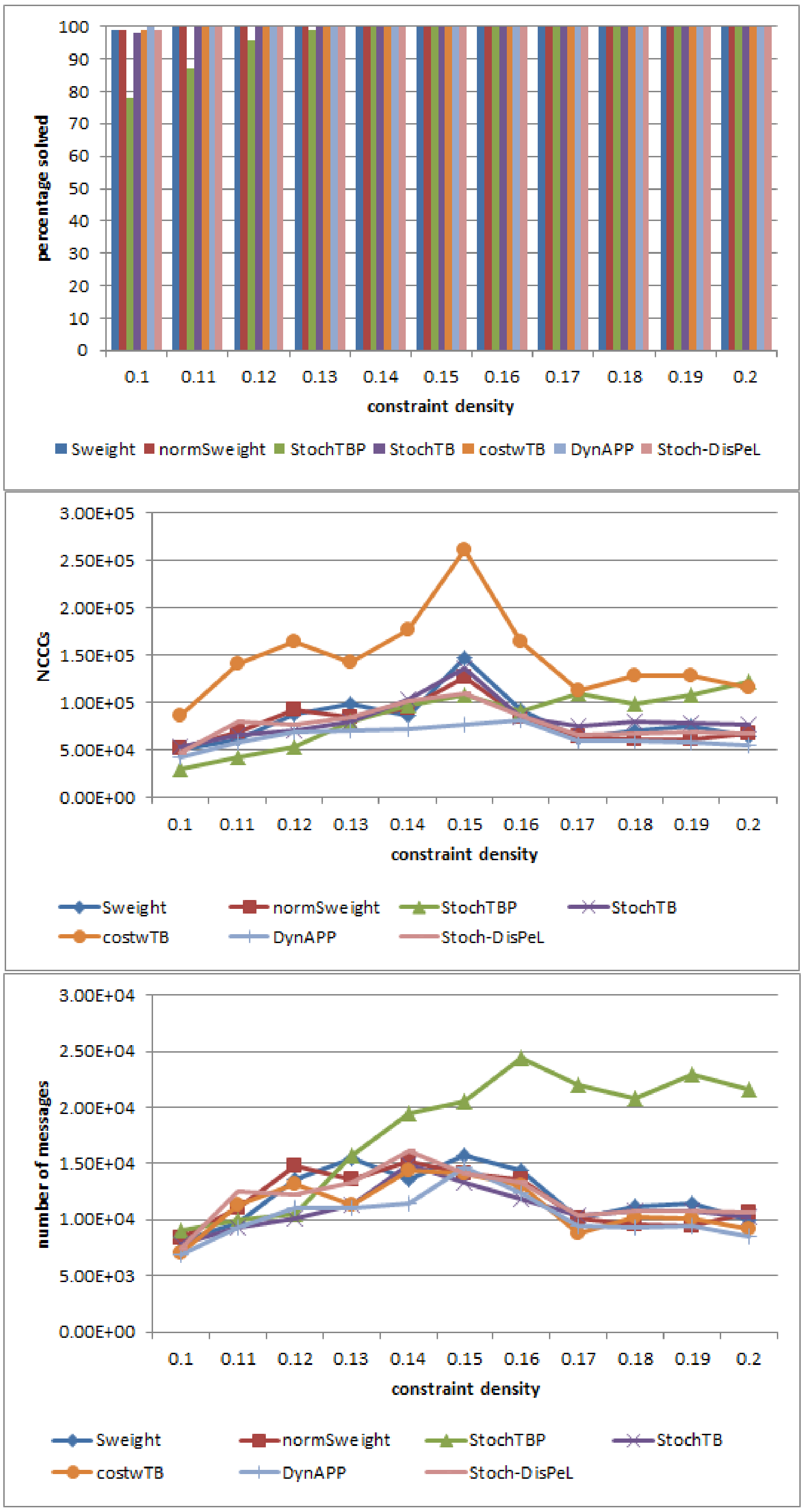


Figure A.4: < n = 200, timeslots = 7, d ∈ {0.1, ..., 0.2}, md = 3 >

A.5 Summary

In this Appendix, we presented heuristics combinations with dynamic agent prioritisation, constraint weights and value penalties. Priorities, constraint weight and value penalties are integer values imposed on parameters of a problem to change how an algorithm searches its solution space to minimise its time to solution. (i) Priorities are attached to agents, initially determined by the alphabetic ordering of the agent identifier and dynamically changed, resulting in a change to the priority of finding a consistent value for the agent's variable; (ii) Weight on constraints make the satisfaction of those constraints more important; and (iii) Penalties make the value undesirable and "hopefully" cause the selection of another value.

We implemented Sweight, Dpenalties, normSweights, StochTBP, StochTB and costwTB, DynAPP and DynCW which are local search algorithms for solving DisCSP having a single variable per agent that differ on how they use the heuristic to calculate the cost function or to break a tie. Empirical evaluations of these algorithms on several problem types with varying difficulty levels show that, at the phase transition where the problems are more difficult, DynAPP, StochTB, StochTBP and costwTB performed better than Stoch-DisPeL on distributed graph colouring problems for both NCCCs and number of messages. Similarly on random DisCSPs, StochTB, Sweight and DynAPP performed better than Stoch-DisPeL. Improvements over Stoch-DisPeL were also seen in StochTB, costwTB and DynAPP on distributed meeting scheduling problems. However, overall, we found DynAPP generally performed best (had the least NCCCs and number of messages) for all problem classes. These improvements in the new algorithms show that the combination strategies are beneficial in distributed local search for single variable per agent settings. In Appendix B we discuss the multi-context search using constraint weights and value penalties.

Appendix B

Multi-Context Search

B.1 Intoduction

In this Appendix, we present a multi-context search with constraint weights and value penalties in local search for DisCSPs having a single variable per agent. The multi-context search aims to improve the exploration of the search space and also to avoid stagnation during search. To solve a given DisCSPs, two local search algorithms for DisCSP having a single variable per agent; SingleDB-wd and Stoch-DisPeL concurrently (refer to Chapter 4, Section 4.3.1 for a description of the algorithms). Each algorithm starts from a different random initialisation in a master-slave architecture. The master algorithm is responsible for finding and returning a solution or terminate if the maximum number of iterations is reached while the task of the slave algorithm is to find and store its best "solution" which are sent to the master algorithm when requested. After some specified number of iterations, the master algorithm sends a message to the slave algorithm asking if a better "solution" (with a lower number of constraint violations) exists and re-initialises its variables with the received assignments from the slave algorithm. Next we describe our implementations in Section B.2 and the empirical evaluation on several problem classes in Section B.3. We summarise the appendix in Section B.4.

B.2 Implementations

We implemented two local search algorithms for DisCSPs having a single variable per Agent with the multi-context search that differ on which algorithm is master or slave.

(a) **StochDisPeL+Restart :** Stoch-DisPeL is the *master algorithm* and SingleDB-wd is the *slave algorithm*.

(b) **SingleDB-wd+Restart :** SingleDB-wd is the *master algorithm* and Stoch-DisPeL is the *slave algorithm*.

Both algorithms are used to solve a given problem concurrently until a termination condition is reached i.e. solution is returned by the *master algorithm* or the maximum number of iterations has been exhausted. Although both algorithms attempt to solve the problem, only the master algorithm is expected to return a solution.

When to Restart : To determine when a *master algorithm* should request for a better solution from a slave, we empirically tried several values. We found that (a) if a small number of iterations is used, the algorithm restarts too often without observing the impact of the restart; (b) if a larger number of iterations is used, the algorithm hardly restarts and when it does restart it yields a worse result; and (c) if a value in the middle (not too small or large) is used, it restarts before stagnation is encountered and also enables the *master algorithm* to improve from the restart assignments received. As a result of the empirical evaluation, we set a parameter *checkIteration* that represents the number of iterations before a restart to the total number of variables in the DisCSP. After every *checkIteration*s number of iterations, the *master algorithm* sends a message to the slave algorithm asking if a better (i.e has less number of constraint violations) solution was found by the *slave algorithm*. If such a solution exists, the *master algorithm* would then reinitialise its variables with the assignments from the slave algorithm. Both algorithms continue the search.

B.3 Empirical Evaluation of the Multi-Context Search Algorithms

This section discusses the performance of the multi-context search algorithms: StochDisPeL+Restart and SingleDB-wd+Restart on a number of problems; **distributed graph colouring problems, random DisCSPs and distributed meeting scheduling problems** (refer to Chapter 3 for the description of the problems). Because StochDisPeL+Restart and SingleDB-wd+Restart are algorithms for DisCSPs where an agent represents a single variable, we exempt sensor network problems from our evaluation due to its inherent nature of having 3 variables per agent. For each problem setting, we run 100 solvable problem instances from which we record and present (i) the percentage of problems solved within the maximum number of iterations; (ii) median of the number of Non-Concurrent Constraint Checks (NCCCs) performed; and (iii) median of the number of messages sent as the measure of efficiency. Although CPU time is not an established measure for DisCSPs (Meisels et al. 2002) and is not reported in this thesis, we measured CPU time and found the results to be consistent with the other evaluation metrics used.

The results are benchmarked with three other local search algorithms for DisCSPs having single variable per agent: **StochDisPeL**, **SingleDB-wd** and **DynAPP**. We excluded DSA (Zhang et al. 2002) due to the fact that it has no explicit mechanism for escaping from local optima, thus, does not perform well where the goal is to satisfy all constraints. Once stuck at local optima, the sideways moves are usually insufficient to push a search out of locally optimal regions. Each algorithm was allowed to run for a maximum of (100 * (n)) iterations where n represents the number of variables (nodes, variables or meetings). Note: In the few cases where not all problems were solved, the effort "wasted" in that problem are not counted towards the median number of messages and median NCCCs, they do however affect the percentage of problems solved.

Results on Distributed Graph Colouring Problems

We present results for 3-colour distributed graph colouring problems, with the number of nodes $n \in \{100, ..., 200\}$ in steps of 10, 3 colours and a degree deg of 4.9 in steps of 0.1 were used. The results are illustrated in Figure B.1. For the number of problems solved, DynAPP and StochDisPeL+Restart solved at least as much as the other algorithms. Similarly on NCCCs and number of messages, DynAPP and StochDisPeL exchanged the least number of messages and NCCCs compared to SingleDB-wd and SingleDB-wd+Restart with SingleDB-wd having the worst performance.

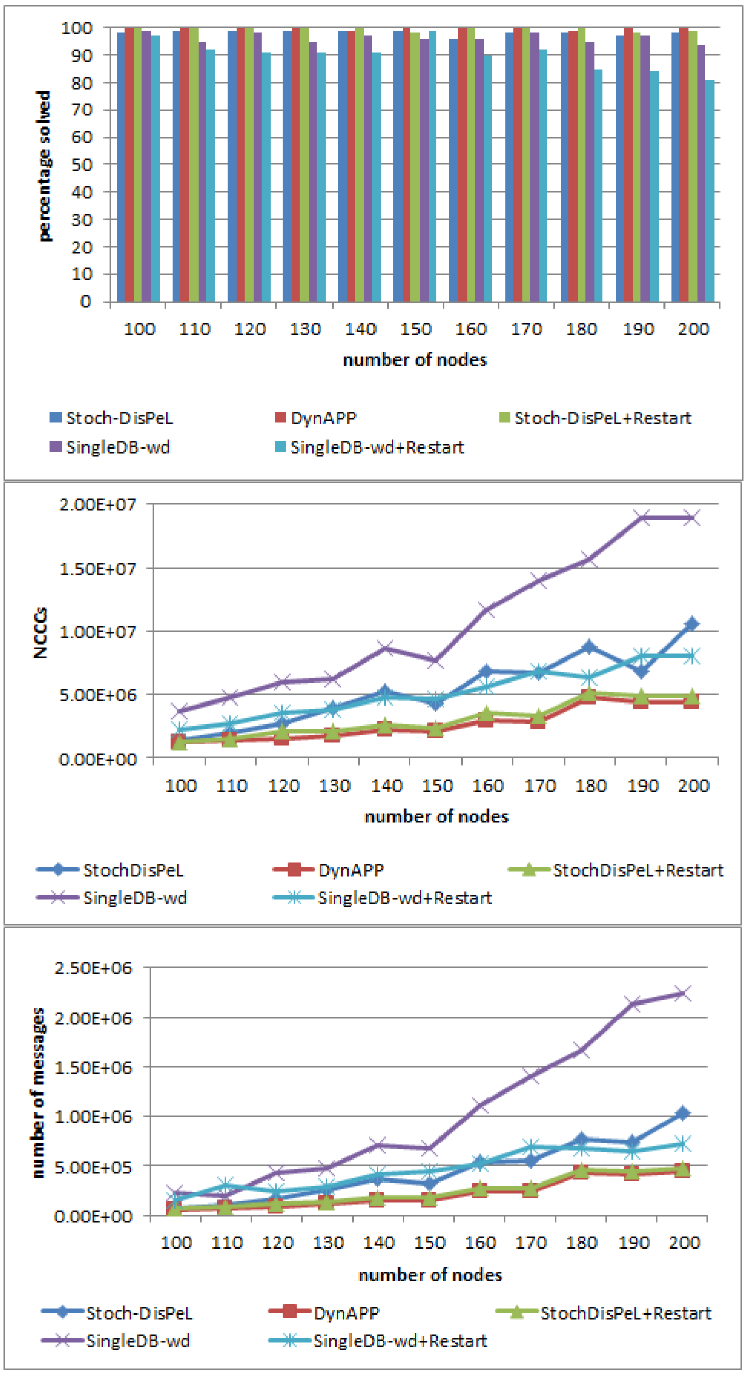


Figure B.1: < n ∈ {100, ..., 200}, |colours| = 3, deg = 4.9 >

Results on Random Distributed Constraint Satisfaction Problems

We generated random problems with different number of variables, a constraint tightness p_2 0.4 i.e. the region of difficult problems and a domain size of 8. We present results where the number of variables $n \in \{80, ..., 200\}$ with constraint density p_1 of 0.2. The results are presented in Figure B.2. SingleDB-wd+Restart solved less than 40% (within the maximum number of iterations) of the problems from n = 140. To highlight the performance of algorithms that solved more problems, the SingleDB-wd results for 140 or more variable were not included in our illustrations of the results as it would obscure the differences in performance for the other algorithms. However, DynAPP solved the most problems. The number of messages and NCCCs increased as the number of variables increases with DynAPP and StochDisPeL+Restart exchanging the least number of messages compared to StochDisPeL, SingleDB-wd and SingleDB-wd+Restart.

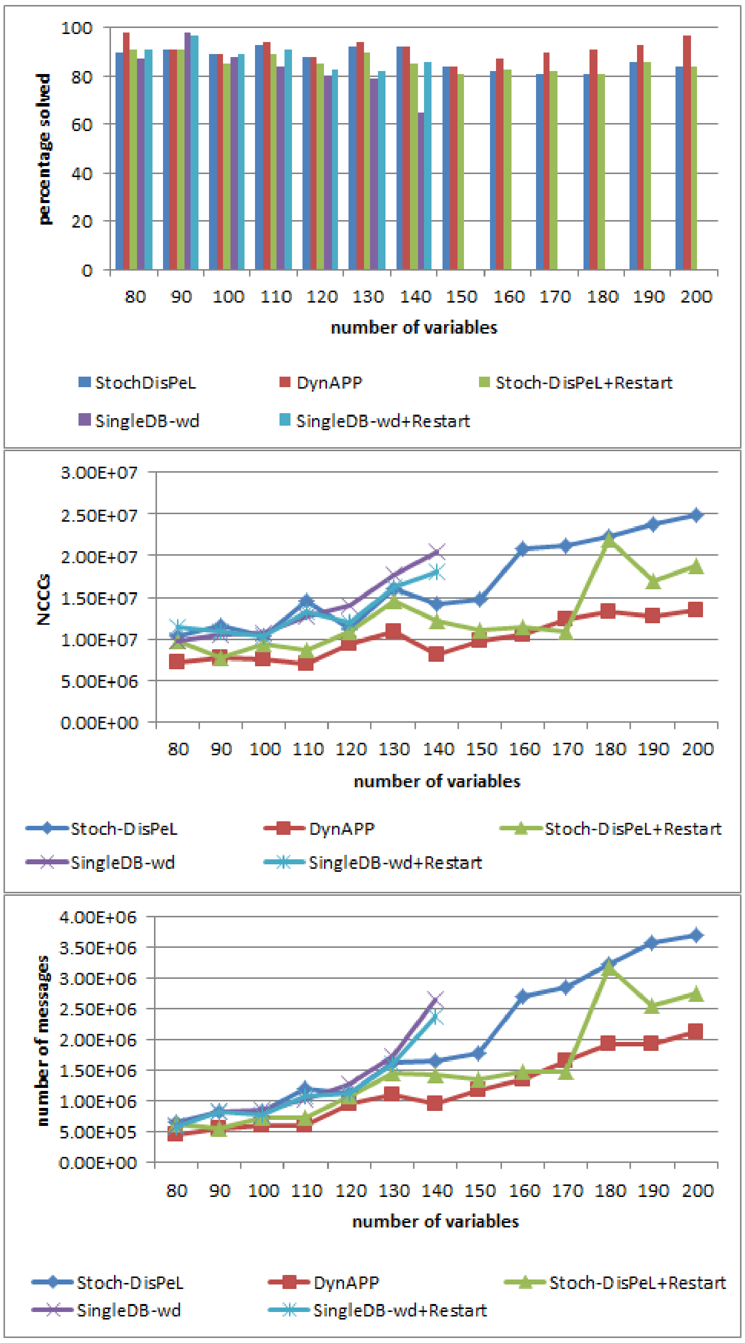


Figure B.2: $< n \in \{80, ..., 200\}, |\text{domain}| = 8, p_1 = 0.2, p_2 = 0.4 >$

Results on Distributed Meeting Scheduling Problems

For distributed meeting scheduling problems, in Figure B.3, we present results for number of meetings m $\in$ {100, ..., 200} in steps of 10, 7 timeslots, maximum possible distance (md) of 3 and a constraint density d of 0.2.

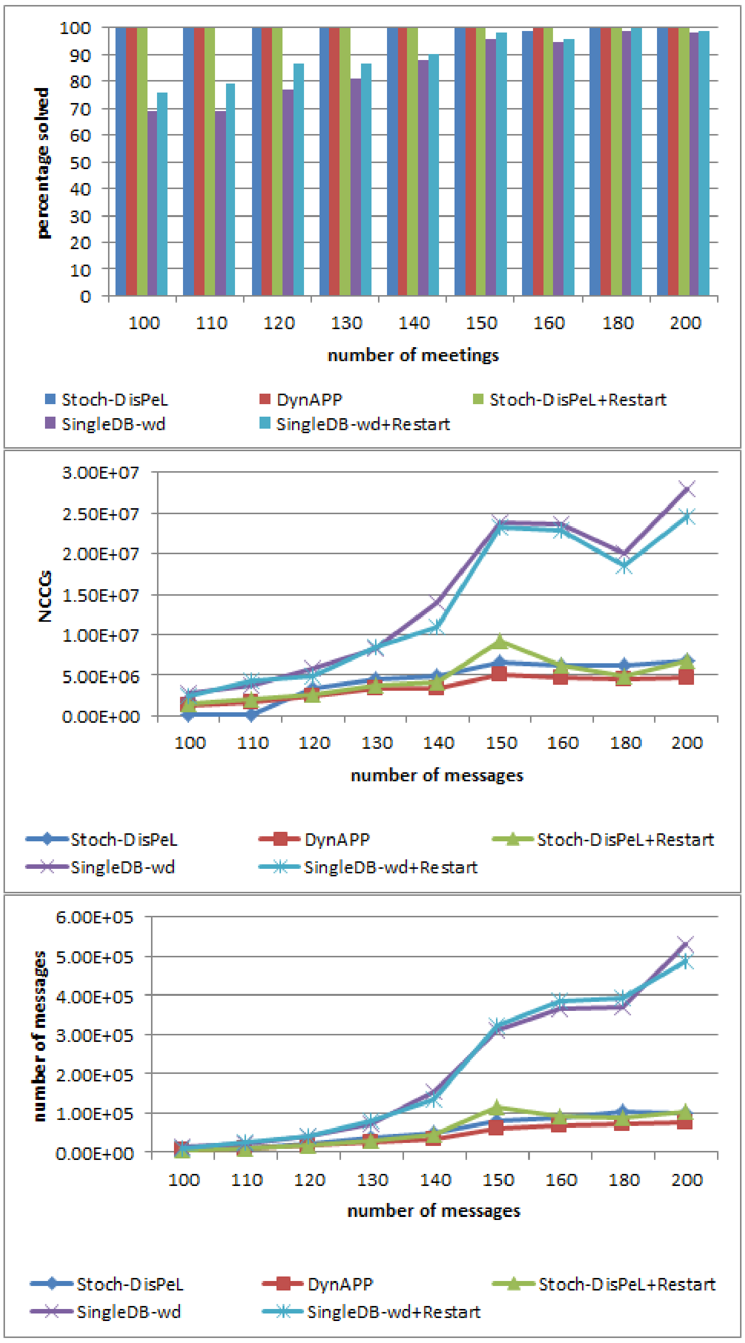


Figure B.3: <n $\in$ {100, ..., 200}, timeslots = 7, d = 0.2, md = 3 >

DynAPP and StochDisPeL+Restart solved all the problems with SingleDB-wd+Restart solving the least number of problems. Initially, all three algorithms performed equally but as the number of meetings increased, the NCCCs and number of messages used increased. DynAPP, StochDisPeL and StochDisPeL+Restart performed closely with DynAPP as the overall best from 150 meetings.

B.4 Summary

In this Appendix, we presented a multi-context search approach to solving DisCSPs with single variable per agent. We use two local search algorithms; Stoch-DisPeL (imposes penalty on values at quasi-local optima) and SingleDB-wd (uses a constraint weight decay mechanism at every iteration) that run concurrently in a master-slave architecture. The *master algorithm* solves the problem and returns a solution if found. However, the *slave algorithm* solves the problem to serve the *master algorithm* by providing assignments to variables used to restart variables in the *master algorithm*. We implemented the multi-context search in StochDisPeL+Restart and SingleDB-wd+Restart local search algorithms for single variable per agent. The results of empirical evaluation on several problem instances showed SingleDB-wd+Restart performed poorly compared to both DynAPP and StochDisPeL+Restart but show improvements over SingleDB-wd. However, StochDisPeL+Restart compared closely to DynAPP in most problem instances but overall, DynAPP still performed best.

Appendix C

Multi-DynAPP

C.1 Intoduction

In the divide and conquer approach for solving DisCSP with CLPs, the structure of the DisCSP is analysed in order to identify groups of directly-related (via an intra-agent constraint) external variables within each agent. These groups are referred to as *compound groups*. A combination of local search and systematic search algorithms are then run concurrently to solve the divided problem (refer to Chapter 6, Section 6.3 for details on the divide and conquer approach). In this Appendix, we present Multi-DynAPP - Dynamic Agent Prioritisation with Penalties for Agents with CLPs. Multi-DynAPP.

In chapter 6, we discussed Multi-DCA, another algorithm that implements the divide and conquer approach and revises Multi-DynAPP. The distributed local search in both Multi-DynAPP and Multi-DCA are an adaptation of Dynamic Agent Prioritisation with Penalties (DynAPP) for compound domains (discussed in Chapter 5). The principal differences between Multi-DCA and Multi-DynAPP are: (i) For the implementation of DynAPP used, penalty messages in Multi-DynAPP are sent to only a subset of an agent's neighbours while in Multi-DCA, penalty messages are sent to all neighbours; and (ii) in Multi-DCA, large compound groups with large number of variables and domain sizes are partitioned into smaller groups. Next, we present preliminary empirical evaluation of Multi-DynAPP on random DisCSPs in Section C.2. We compare Multi-DynAPP and Multi-DCA in Section C.3 and summarise in Section C.4.

C.2 Empirical Evaluation of Multi-DynAPP

Our implementation of Multi-DynAPP was evaluated on a wide variety of random DisCSPs (refer to Chapter 3 for the description of random problems) with different characteristics (each characteristic is described in the appropriate section). For each problem setting, we run 100 solvable problem instances from which we record and present: (i) the percentage of problems solved within the maximum number of iterations; (ii) median of the number of Non-Concurrent Constraint Checks (NCCCs) performed; and (iii) median of the number of messages sent to measure efficiency. Although CPU time is not an established measure for DisCSPs (Meisels et al. 2002) and is not reported in this thesis, we measured CPU time and found the results to be consistent with the other evaluation metrics used. To generate naturally distributed DisCSPs (except for distributed sensor network problems) with complex local problems, the problems considered contained between 70(30) and 80(20) ratio of intra(inter) agent constraints and the exact ratio used in the result presented is specified in the experiments. The ratio of variables within the complex local problem are 70(30) internal (external) variables. Each algorithm was allowed to run for a maximum of (100 * (n)) iterations where n represents the number of variables (nodes, variables, meetings or sensors).

The results are benchmarked with two local search algorithms for DisCSP with CLPs: **Multi-DisPeL** and **DisBO-wd**. Multi-DisPeL (value penalty) and DisBO-wd (constraint weights) are local search algorithms (refer to Chapter 4, Section 4.3 for the description of algorithms). We obtained the implementations of Multi-DisPeL, DisBO-wd and Multi-Hyb-Pen from their authors.

Note: The following algorithms were not considered; (i) DCDCOP (Khanna et al. 2009) was not considered for evaluation as the pseudocode available is insufficient to implement the algorithm and it is designed for DCOP; (ii) Burke's work (Burke & Brown 2006a), (Burke & Brown 2006b) concentrated on efficiency in handling CLPs and there is no overall algorithm; and (iii) ADOPT (Modi et al. 2005) is also designed for distributed constraint optimization. In the few cases where not all problems were solved, the effort "wasted" in that problem are not counted towards the median number of messages and median NCCCs, they do however affect the percentage of problems solved. Multi-Hyb-Pen is a complete algorithm and it solved all problems as expected.

We present results for experiments where we vary the following problem parameters: (i) constraint tightness; (ii) number of variables; (iii) number of agents; and (iv) domain size.

Tightness	**DisBO-wd**	**Multi-DisPeL**	**Multi-DynAPP**
	Percentage Solved		
0.2	100	100	100
0.25	100	100	100
0.3	100	100	100
0.35	100	100	100
0.4	99	100	100
0.45	**99**	**100**	**99**
0.5	95	**98**	95
	NCCCs		
0.2	**5,676**	10,286	19,009
0.25	**7,997**	12,657	21,913
0.3	**10,390**	15,067	35,679
0.35	**10,280**	22,943	36,065
0.4	**19,534**	32,533	61,504
0.45	**31,170**	61,217	62,842
0.5	**39,209**	89,469	67,513
	Messages		
0.2	50	**20**	**20**
0.25	25	**20**	**20**
0.3	35	**24**	30
0.35	45	**40**	**40**
0.4	75	**60**	**60**
0.45	125	140	**104**
0.5	140	164	**100**

Table C.1: $< n = 80, |domain| = 8, |agents| = 5, p_1 = 0.15, p_2 \in \{0.2, ..., 0.5\} >$

- **Varying the Constraint Tightness :** For experiments with varying constraint tightness, 80 variables were used and constraint tightness $p_2 \in \{0.2, ..., 0.5\}$ in steps of 0.05. The experiments used a domain size of 8, 5 agents and constraint density p_1 of 0.15. As seen in Table C.1 all the algorithms solved at least 90% of the problems and except for $p_2 = 0.3$, agents in Multi-DisPeL and Multi-DynAPP exchanged the least messages in solving the problems. On the other hand, DisBO-wd used the least NCCCs.

- **Varying the Number of Variables :** For these experiments, we conduct two sets of experiments by using (i) a fixed number of variables; or (ii) a varying number of variables within an agent. This is to determine the performance of increasing number of variables on agents, by sharing the problem on a fixed or varying number of agents.

  (i) **The number of variables per agent varies for each problem**

  In Table C.2, we present results for problems with 100 to 180 variables in steps of 10, with a domain size of 7, a constraint density p_1 of 0.15, a constraint tightness p_2 of 0.5 and 10 agents for each problem size. The number of variables per agent varies according to the problem size, i.e. it is 10 for 100 variables, 11 for 110 variables and so on. We present results for two different ratios of intra-inter constraints. In both cases, Multi-DynAPP and Multi-DisPeL performed closely with the least number of messages except for 100 variables in the 70(30) intra(inter) constraints. However, Multi-DynAPP used the least NCCCs. For the percentage of problems solved, Multi-DynAPP solved at least as much as Multi-DisPeL and DisBO-wd.

(ii) **The number of variables per agent is fixed**

In Figure (C.1), we present results for problems with 50 to 150 variables in steps of 10, a domain size of 7, a constraint density p_1 of 0.15, a constraint tightness p_2 of 0.5, the number of variables per agent was fixed at 10, thus, the number of agents varies for each problem size, i.e. it is 5 agents for 50 variables, 6 agents for 60 variables and so on. Agents in Multi-DynAPP generally exchanged the least number of messages and performed less computational effort (NCCCs) than Multi-DisPeL and DisBO-wd to solve a problem. For 70 variables, all 3 algorithms exchanged similar number of messages and for 150 variables the number of messages dropped drastically for the 3 algorithms. This could be due to the increased number of variables per agent with a fixed constraint density or constraint tightness possibly making the problems easier. All the algorithms solved all problems within the given number of iterations.

- **Varying the Number of Agents :** A number of experiments were conducted where the number of agents was varied. In these experiments, 80 variables were used and agents $\in \{4, 5, 8, 10\}$ such that each agent has an equal number of variables i.e. 20, 16, 10 and 8 respectively. As seen in Table C.3, more problems were solved by Multi-DynAPP, Multi-DisPeL and DisBO-wd as the number of agents increased. Overall, Multi-DynAPP significantly reduced the number of messages although it solved less problems when 4 and 5 agents were used.

- **Varying the Domain Size :** A number of experiments were conducted for this experiment to determine the performance of Multi-DynAPP on domain size. 80 variables were used and domain size $\in \{3, ..., 10\}$ in steps of 1. 5 agents were used with a constraint density p_1 of 0.15 and a constraint tightness p_2 of 0.5. The results in Table C.4 shows the % of problems solved decreased for Multi-DynAPP, Multi-DisPeL and DisBO-wd as the domain size increased. The algorithms alternated for the least NCCCs while Multi-DynAPP generally performed worst, solving fewer problems compared

	80:20 intra:inter constraints			**70:30 intra:inter constraints**		
Var	**DisBO-wd**	**Multi-DisPeL**	**Multi-DynAPP**	**DisBO-wd**	**Multi-DisPeL**	**Multi-DynAPP**
	Percentage Solved			**Percentage Solved**		
100	100	100	100	100	100	100
110	100	100	100	100	100	100
120	100	100	100	100	100	100
130	98	100	99	100	100	100
140	98	100	100	100	100	100
150	97	99	98	100	100	100
160	95	99	99	99	100	100
170	95	98	99	97	99	100
180	95	99	98	97	100	100
	NCCCs			**NCCCs**		
100	181,531	299,276	**68,739**	160,288	286,532	**67,118**
110	181,004	353,207	**62,641**	226,140	287,687	**92,193**
120	253,300	348,270	**69,807**	265,406	357,803	**62,438**
130	269,511	391,496	**52,783**	250,393	365,550	**81,883**
140	307,422	397,081	**85,131**	303,809	397,297	**114,577**
150	298,383	456,885	**128,673**	289,008	472,973	**143,193**
160	**376,914**	496,032	492,918	**357,885**	545,000	413,330
170	**422,208**	580,438	738,619	**473,529**	559,210	793,660
180	**481,531**	649,096	768,047	**450,285**	650,397	757,168
	Messages			**Messages**		
100	380	544	**220**	330	634	270
110	300	477	**232**	330	351	**180**
120	274	300	**240**	320	346	**180**
130	260	**180**	**180**	230	**180**	**177**
140	240	211	**180**	250	180	**157**
150	210	**180**	**180**	220	180	**140**
160	220	190	**180**	190	**110**	**109**
170	270	**110**	**108**	260	**110**	**120**
180	250	**100**	**100**	260	**105**	**100**

Table C.2: $< n \in \{100, ..., 180\}$, |domain| = 7, |agents| = 10, $p_1 = 0.15$, $p_2 = 0.5 >$

to Multi-DisPeL as the domain size increases from 7 to 10. Note, for a domain size of 10, DisBO-wd solved only 45% of the problems and this is indicated by an asterix.

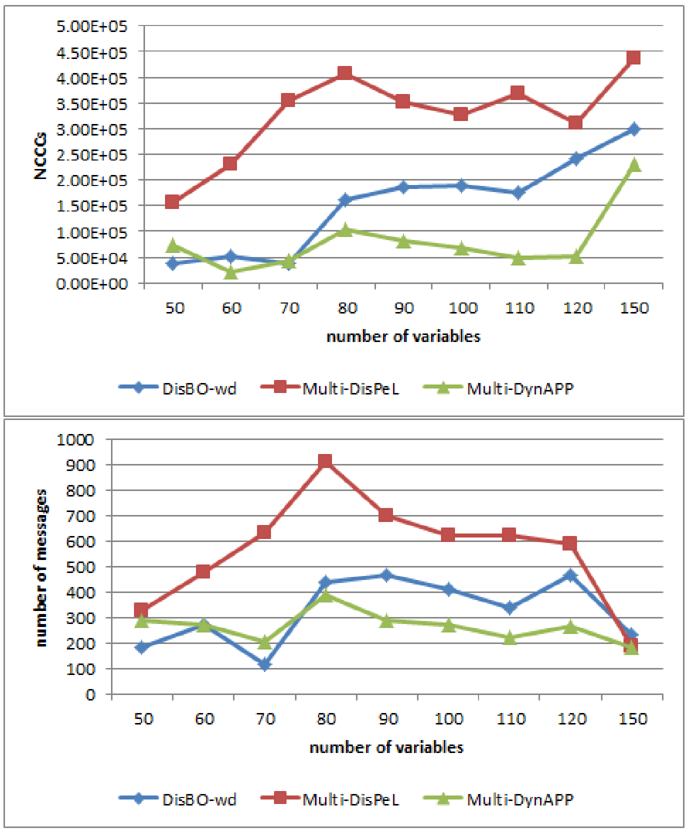


Figure C.1: $< n \in \{50, ..., 150\}, |\text{domain}| = 7, |\text{agents}| \in \{5, ..., 15\}, p_1 = 0.15, p_2 = 0.5 >$

Agents	**DisBO-wd**	**Multi-DisPeL**	**Multi-DynAPP**
		Percentage Solved	
4	75	98	95
5	75	98	95
8	100	100	100
10	100	100	100
		NCCCs	
4	**138,246**	461,399	908,409
5	**39,209**	89,469	67,513
8	407,364	161,893	**105,164**
10	56,780	265,642	**25,577**
		Messages	
4	159	168	**30**
5	140	164	**100**
8	910	440	**389**
10	328	900	**225**

Table C.3: $< n = 80, |\text{domain}| = 8, |\text{agents}| \in \{4, 5, 8, 10\}, p_1 = 0.15, p_2 = 0.5 >$

Domain Size	**DisBO-wd**	**Multi-DisPeL**	**Multi-DynAPP**
		Percentage Solved	
3	100	100	100
4	100	100	100
5	100	100	100
6	100	100	100
7	88	99	98
8	76	99	95
9	70	100	95
10	45	90	90
		NCCCs	
3	13,133	28,945	**4,076**
4	22,428	50,846	**12,419**
5	54,980	90,314	**53,715**
6	**187,496**	76,402	80,532
7	**93,407**	268,124	230,879
8	**120,721**	460,910	322,958
9	**153,380**	659,970	864,296
10	*	**791,407**	1,191,020
		Messages	
3	**50**	**50**	**50**
4	60	**48**	**50**
5	95	**84**	**80**
6	135	132	**80**
7	160	168	**130**
8	185	272	**100**
9	215	386	**170**
10	*	336	**130**

Table C.4: $<$ n = 80, $|\text{domain}| \in \{3, ..., 10\}$, $|\text{agents}| = 5$, $p_1 = 0.15$, $p_2 = 0.5 >$

C.3 Comparing Multi-DynAPP and Multi-DCA

The results of the empirical evaluation of Multi-DynAPP on random DisCSPs shows some general improvement mainly in computation costs when compared to DisBO-wd and Multi-DisPeL. However, Multi-DynAPP did not perform well on problems where variables have large domain sizes. In such a setting, the *combination solutions* becomes larger thus, resulting in domain values having several similar assignments for most of the variables in the *compound groups*. To address this, we introduce partitioning of *compound groups* with large number of variables and large domain sizes in Multi-DCA.

We compared Multi-DCA and Multi-DynAPP in an empirical evaluation on random DisCSP with the following settings:

- A fixed number of variables and varying domain sizes:

  The results presented in Table C.5 are for 150 variables (n), a constraint tightness p_2 of 0.5, a constraint density p_1 of 0.2 and we vary domain size $\in \{3, ..., 10\}$ in steps of 1. Initially, the number of messages for both Multi-DCA and Multi-DynAPP were the same, but, as the size of the domain increased, Multi-DCA generally used less. Overall, on the NCCCs and on percentage of problems solved, Multi-DCA outperformed Multi-DynAPP.

- A fixed domain sizes and varying number of variables

  In Table C.6, we present results for number of variables (n) $\in \{100, ..., 170\}$ in steps of 10, a constraint tightness p_2 of 0.5, a constraint density p_1 of 0.2 and a domain size of 6. Similar to the result presented on varying domain sizes, initially, the number of messages for both Multi-DCA and Multi-DynAPP were the same, but, as the size of the number of variables increases from 120, Multi-DCA used less. However, overall on the NCCCs and on percentage of problems solved, Multi-DCA generally outperformed Multi-DynAPP across all problem sizes.

Domain size	**Multi-DynAPP**	**Multi-DCA**
	Percentage Solved	
3	100	100
4	100	100
5	97	99
6	95	99
7	95	100
8	96	99
9	97	100
10	95	98
	NCCCs	
3	**7,780**	**7,926**
4	15,616	**12,482**
5	74,389	**46,299**
6	174,080	**136,299**
7	569,021	**269,173**
8	1,034,137	**495,836**
9	3,799,998	**1,477,760**
10	28,461,096	**9,773,190**
	Messages	
3	40	40
4	40	40
5	40	40
6	40	40
7	60	60
8	80	**60**
9	100	**80**
10	120	**80**

Table C.5: $< n = 150, |\text{domain}| \in \{3, ..., 10\}, |\text{agents}| = 10, p_1 = 0.2, p_2 = 0.5 >$

No. of Variables	**Multi-DynAPP**	**Multi-DCA**
	Percentage Solved	
100	100	100
110	98	100
120	97	100
130	95	99
140	95	98
150	96	99
160	97	99
170	95	98
	NCCCs	
100	**13,770**	**13,121**
110	14,016	**14,002**
120	24,389	**18,200**
130	51,087	**36,299**
140	117,021	**80,073**
150	284,257	**153,801**
160	722,900	**450,000**
170	1,091,004	**673,190**
	Messages	
100	**80**	**74**
110	**102**	**100**
120	**120**	**120**
130	144	**128**
140	161	**130**
150	180	**150**
160	200	**160**
170	220	**180**

Table C.6: $< n \in \{100, ..., 170\}$, $|\text{domain}| = 6$, $|\text{agents}| = 10$, $p_1 = 0.2$, $p_2 = 0.5 >$

C.4 Summary

In this Appendix, we presented Multi-DynAPP, a divide and conquer approach for solving DisCSPs with CLPs. The approach combines the following strategies: (i) both systematic and local searches; (ii) both centralised and distributed searches; and (iii) a modified compilation reformulation strategy for a more effective resolution of DisCSPs while also exploiting concurrency and problem structure. A complete discussion of the divide and conquer approach was presented in Chapter 6, Section 6.2.

We presented results of an empirical evaluation of Multi-DynAPP on a number of random DisCSPs with different characteristics. From the results, we discovered that Multi-DynAPP performed best when the number of variables and domain of the variable in a *compound group* are small. As the number of variables and domain size increases, the possible number of *combination solutions* increases. To address large compound domains, we introduce the partitioning of large compound groups. Multi-DCA (discussed in Chapter 6),is an implementation of the divide and conquer approach with partitioning. Thus, the main difference between Multi-DynAPP and Multi-DCA is the partitioning of large compound groups into smaller groups as a strategy for dealing with potentially large combination solutions. Also, the implementation of DynAPP used, penalty messages in Multi-DynAPP are sent to only a subset of an agent's neighbours while in Multi-DCA, penalty messages are sent to all neighbours.