PARTICLE SWARM OPTIMIZATION FOR SOLVING CONSTRAINT SATISFACTION PROBLEMS

by

I-Ling Lin

B. Sc., Simon Fraser University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Interactive Arts and Technology

© I-Ling Lin 2005
SIMON FRASER UNIVERSITY
Fall 2005

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: I-Ling Lin

Degree: Master of Science

Title of thesis: Particle Swarm Optimization for Solving Constraint Satisfaction Problems

Examining Committee:

____

Dr. Rob Woodbury, Professor,
School of Interactive Arts and Technology
Simon Fraser University
Chair

____

Dr. Marek Hatala, Assistant Professor,
School of Interactive Arts and Technology
Simon Fraser University
Senior Supervisor

____

Dr. Toby Donaldson, Lecturer, Computing Science
Simon Fraser University
Supervisor

____

Dr. Vive Kumar, External Examiner,
Assistant Professor,
School of Interactive Arts and Technology
Simon Fraser University

Date Approved: Aug 30/2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.\

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

W. A. C. Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

This research presents the design and evaluation of a variety of new constraint-solving algorithms based on the particle swarm optimization (PSO) paradigm. Constraint satisfaction problems (CSPs) can be applied to many practical problems but they are in general NP-hard, so developing new algorithms has been a major research challenge. PSO is a relatively new approach to AI problem solving and has just begun to be applied to CSPs. This research modifies and extends the traditional PSOs to solve n-ary CSPs. These new particle swarm algorithms are tested on practical configuration problems and the traditional n-queens problems. The effectiveness and efficiency of the new algorithms are experimentally compared to the traditional PSOs. The performance of the individual algorithms is also assessed. The algorithms that combine zigzagging particles and repair-based CSP-solving methods perform best among the algorithms studied.

To my parents, my homestay mother and Linda...

Acknowledgments

I would like to thank those people who have supported me during the course of this research. Without their help, I would not be able to complete this thesis.

First, I would like to thank Dr. Toby Donaldson for inspiring me, supervising me and sponsoring me. He has been available since my undergraduate studies in Computing Science at Simon Fraser University. He has guided and encouraged me throughout this research.

Also, I would like to thank Dr. Marek Hatala for being my senior supervisor. He advised and helped me through writing and defending the thesis.

Dr. Rob Woodbury assisted me to evaluate my research and experiments, and I would like to acknowledge him and owe my thanks to him.

I am grateful to Dr. Vive Kumar for being my external examiner as well.

I would also like to thank our graduate program assistants and advisors, particularly Allison Neil and Joyce Black for helping me throughout these years. In addition, I would like to thank the staff in Academic Computing Service (ACS) at Surrey for providing the computer facilities for my experiments.

I should also appreciate the Faculty of Applied Sciences at Simon Fraser University, the School of Interactive Arts and Technology (SIAT) at Simon Fraser University, the Donor of the Global (West) Wholesalers Ltd. Graduate Bursary in Expert Systems and the Donor of the Ralph M. Howatt Family Graduate Scholarship in Expert Systems for offering me scholarships, fellowships and bursaries.

I also want to thank my parents for their everlasting love, patience and financial support all these years. Moreover, I would like to thank Ms. Janice Fox, my homestay mother who has been taking care of me since I came to Vancouver in 1995. Lastly, I would like to acknowledge and thank my roommate, Linda Liu, who helped me gather experimental data, drove me to the labs on numerous weekends and encouraged me all the time.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgments v

Contents vi

List of Tables x

List of Figures xiii

1 Introduction 1

1.1 Motivation 1

1.2 Research Goal 2

1.3 Thesis Overview 2

2 Constraint Satisfaction Problems 3

2.1 Definitions 4

2.2 Examples 6

2.2.1 A warm-up example 6

2.2.2 Pythagorean triple example 6

2.2.3 8-queens problem 7

2.2.4 send-more-money Puzzle 8

2.2.5 Graph colouring 9

2.3 Problem Solving Techniques . . . 10
2.3.1 Basic search algorithms . . . 10
2.3.2 Problem reduction techniques . . . 12
2.3.3 Strategic search and heuristics . . . 13
2.3.4 Stochastic search . . . 15
2.3.5 Evolutionary Computing . . . 16
2.3.6 Summary of algorithms . . . 17
2.4 CSP Frameworks . . . 17
2.4.1 Arithmetic CSPs in Python . . . 18

3 Particle Swarm Optimization **20**
3.1 Introduction . . . 20
3.2 Swarm Intelligence . . . 21
3.2.1 Ant Colony Optimization . . . 21
3.3 Traditional PSO . . . 23
3.3.1 Definitions . . . 23
3.3.2 Continuous PSO . . . 27
3.3.3 Discrete (Binary) PSO . . . 30
3.4 Solving Problems with PSOs . . . 31
3.4.1 Research problems and applications . . . 31
3.4.2 Strengths . . . 31
3.4.3 Weaknesses . . . 32
3.5 Solving binary CSPs . . . 32
3.5.1 Schoofs and Naudts' operators . . . 33
3.5.2 Parameters used in Schoofs and Naudts' PSO . . . 34
3.6 The Research Goal: PSOs for Solving n-ary CSPs . . . 34

4 Particle Swarm Optimization for Solving CSPs **36**
4.1 CSP Representation in Particle Swarm . . . 36
4.1.1 Connecting CSPs and PSO . . . 37
4.1.2 Handling constraints . . . 38
4.2 PSO Algorithms for Constraint Satisfaction . . . 39
4.2.1 Generic PSO . . . 39
4.2.2 Strategic PSOs . . . 45

4.2.3 Neighbourhood structures . . . 51
4.2.4 Summary of particle swarm algorithms . . . 51
4.3 Application Problem—PC Configuration . . . 53
4.3.1 Introduction . . . 53
4.3.2 Modelling a PC configuration problem in Python CSP Framework . . 53
4.3.3 PC configuration test problems . . . 61

5 Experiment and Evaluation **63**
5.1 Introduction . . . 63
5.2 Experiment Setup . . . 64
5.2.1 Test algorithms . . . 64
5.2.2 Test problems . . . 70
5.2.3 Comparison measures . . . 73
5.3 Experiments . . . 76
5.3.1 Runs . . . 76
5.3.2 Experimental facilities . . . 76
5.3.3 Programming issues . . . 78
5.4 Experimental results . . . 78
5.4.1 Effectiveness . . . 79
5.4.2 Efficiency . . . 92
5.5 Discussion and Answers . . . 105
5.5.1 Can we extend Schoofs and Naudts' PSO to solve general n-ary integer CSPs effectively? . . . 105
5.5.2 How can we modify the traditional PSOs to solve n-ary integer CSPs? How do the algorithms extending the traditional PSOs compare with Schoofs and Naudts' PSO? . . . 107

6 Conclusion **110**
6.1 Summary of the Research Results . . . 111
6.2 Future Work . . . 112

A Algorithms and Examples **114**
A.1 CSP Examples in Python CSP Framework . . . 114
A.2 Swarm Algorithms . . . 117

A.3 Particle Swarm Algorithms for Solving CSPs 120
A.3.1 Local depth-first search: genericDFS 120

B PC Configuration Test Problems **124**
B.1 Formulation I 124
B.1.1 The variables and the domains 124
B.1.2 The constraints 131
B.1.3 Description for Formulation I test problems 131
B.2 Formulation II 133
B.2.1 The variables and the domains 133
B.2.2 The constraints 140
B.2.3 Description for Formulation II test problems 147

C Experimental Setup and Evaluation Data **154**
C.1 Parameter Settings for Exploration Phase 154
C.2 Parameter Settings for Comparison Phase 156
C.3 Figures 159

Bibliography **176**

List of Tables

2.1 A summary of algorithms 17

4.1 This table summarizes the PSO algorithms developed in this research. 52
4.2 Sample CPUs for var_{cpu} and their integer representation. 56
4.3 Sample constraints on var_{cpu}, var_{ram} and var_{mb} under Formulation I. 56
4.4 Four n-ary constraints are added to the Python CSP Framework for Formulation I. 57
4.5 Sample values of CPU specifications and the enumerated domain. 58
4.6 Sample CPUs in good tuples, and the entire list represents "GOODcpu" constraint. 60
4.7 Sample PC connection constraints in Formulation II. 60
4.8 Sample PC user constraints in Formulation II. 61
4.9 PC configuration problems for Formulation I. 62
4.10 PC configuration problems for Formulation II. 62

5.1 The five classes of particle swarm algorithms in this research. 64
5.2 The PSO algorithms used in the Exploration phase. 65
5.3 Parameters used in the Exploration phase 66
5.4 PSO algorithms used in the Comparison phase. 69
5.5 PC configuration problems in the Exploration phase. 71
5.6 n-queens problems in the Exploration phase. 71
5.7 PC configuration problems in the Comparison Phase. 72
5.8 n-queens problems in the all_diff phase. 73
5.9 The systems used in the three-phase experiment. 77
5.10 The success rate of PSO models from the Comparison phase. 84

5.11 The success rate of PSO algorithms from the Comparison phase. 85
5.12 The SR of PSO algorithms from the all_diff phase. 90
5.13 The average run time and the number of consistency checks of *binaryZigzagHop*-distance. 93

B.1 Sample CPUs for var_{cpu} 124
B.2 Sample RAMs for var_{ram} 125
B.3 Sample motherboards for var_{mb} 125
B.4 Sample VGAs for var_{vga} 125
B.5 Sample sound cards for var_{snd} 126
B.6 Sample NICs for var_{nic} 126
B.7 Sample floppy drives for var_{fdd} 126
B.8 Sample hard drives for var_{hdd} 127
B.9 Sample CD-ROMs for var_{cd} 127
B.10 Sample power supplies for var_{power} 128
B.11 Sample casings for var_{tower} 128
B.12 Sample mice for var_{mouse} 128
B.13 Sample monitors for var_{scr} 129
B.14 Sample printers for var_{prt} 130
B.15 Sample keyboards for var_{kb} 130
B.16 PC connection constraints in Formulation I. 131
B.17 Sample values of CPU specifications and the enumerated domain. 133
B.18 Sample values of RAM specifications and the enumerated domain. 134
B.19 Sample values of motherboard specifications and the enumerated domain. . . 134
B.20 Sample values of VGA specifications and the enumerated domain. 135
B.21 Sample values of sound card specifications and the enumerated domain. . . . 135
B.22 Sample values of NIC specifications and the enumerated domain. 136
B.23 Sample values of floppy drive specifications and the enumerated domain. . . . 136
B.24 Sample values of hard drive specifications and the enumerated domain. 137
B.25 Sample values of CD-ROM specifications and the enumerated domain. 137
B.26 Sample values of power supply specifications and the enumerated domain. . . 138
B.27 Sample values of tower case specifications and the enumerated domain. 139
B.28 Sample CPUs in good tuples–component constraint "GOODcpu". 140

B.29 Sample RAMs in good tuples–component constraint “GOODram”. 140
B.30 Sample motherboards in good tuples–component constraint “GOODmb”. . . 141
B.31 Sample VGAs in good tuples–component constraint “GOODvga”. 142
B.32 Sample sound cards in good tuples–component constraint “GOODsnd”. . . . 142
B.33 Sample network cards in good tuples–component constraint “GOODnic”. . . 143
B.34 Sample floppy drives in good tuples–component constraint “GOODfdd”. . . . 143
B.35 Sample hard drives in good tuples–component constraint “GOODhdd”. . . . 144
B.36 Sample CD-ROM drives in good tuples–component constraint “GOODcd”. . 144
B.37 Sample power supplies in good tuples–component constraint “GOODpower”. 145
B.38 Sample tower cases in good tuples–component constraint “GOODtower”. . . 145
B.39 Sample PC connection constraints in Formulation II. 146
B.40 Sample PC user constraints in Formulation II. 147
B.41 CSP variables of problem set 20. 147
B.42 CSP variables of problem set 21. 148
B.43 CSP variables of problem set 22. 149
B.44 CSP variables of problem set 23. 149
B.45 CSP variables of problem set 24. 150
B.46 CSP variables of problem set 25. 151
B.47 CSP variables of problem set 26. 153

C.1 Parameter settings used in Exploration phase 154
C.2 Parameter settings used in Comparison phase 156
C.3 The *partial* success rate of PSO algorithms: problem set 10 from the Comparison phase. 159
C.4 The success rate of the PSO parameter settings. 159

List of Figures

2.1 A Pythagorean triple example in Section 2.2.2. 7
2.2 Queen positions in a 4-queens problem. 8
2.3 A sample solution of the 8-queens problem. 8
2.4 A send-more-money puzzle and its solution. 9
2.5 A sample graph colouring problem with 3 colours. 10
2.6 A constraint graph of the graph colouring problem in Section 2.2.5. 13
2.7 A Sample Run of the graph colouring problem in Python CSP framework. . . 19

3.1 Particle swarm (population = 10) in a 2-dimensional space. 23
3.2 A position-velocity relation in a 2-dimensional space. 24
3.3 A global swarm vs. local neighbourhoods [25]. 25
3.4 Simple neighbourhood topologies (population = 5) [51, 25]. 25

4.1 A particle position x_i in the CSP context is a complete assignment. 37
4.2 A particle velocity v_i changes CSP variable assignment. 38
4.3 A pseudocode segment describes the change done for the Continuous PSO. . 40
4.4 A position adjustment on the jth-axis when the element x_{ij} goes out of domain D_j. 40
4.5 Schoofs and Naudts' PSO [90]. It is named as *bcspPSO* and serves as the foundation of all algorithms derived from BCSP model in this research. . . . 44
4.6 A particle moves in a 2-dimensional space with 2 different styles. 46
4.7 Exchanging partner is taking actions. 50
4.8 The CSP variables of a PC configuration problem under Formulation I. . . . 55
4.9 The CSP variables of a PC configuration problem under Formulation II. . . . 59

5.1 The success rate of PSO models from the Comparison phase. 80

5.2 The mean evaluation value of PSO models from problem set 25. 83
5.3 The mean evaluation value of PSO algorithms from problem set 25. 86
5.4 The success rate of populations from the Comparison phase. 87
5.5 The success rate of *pop_rate*: hop and zigzagHop algorithms from the Comparison phase. 87
5.6 The mean evaluation value of *pop_rate* from problem set 25. 88
5.7 The mean evaluation value of PSO algorithms from the Comparison phase. . 89
5.8 The success rate of PSO models from the all_diff phase. 93
5.9 The mean run time of PSO models from problem set 25. 97
5.10 The mean run time of PSO algorithms from problem set 25. 98
5.11 The mean run time of PSO algorithms from problem set 25. 99
5.12 The mean number of consistency checks of PSO models from problem set 25. 100
5.13 The mean number of consistency checks of PSO algorithms from problem set 25. 101
5.14 The mean number of consistency checks of PSO algorithms from the Comparison phase. 102
5.15 The mean run time of PSO *pop_rate*: problem set 25 from the Comparison phase. 103
5.16 Mean number of consistency checks of *pop_rate*: problem set 25 from Comparison phase. 104

A.1 The warm-up example of Section 2.2.1 in the Python CSP framework. 114
A.2 The Pythagorean triple example of Section 2.2.2 in the Python CSP framework. 114
A.3 8-Queens problem of Section 2.2.3 in the Python CSP framework. 115
A.4 The send-more-money puzzle of Section 2.2.4 in the Python CSP framework. 115
A.5 The sample graph colouring problem of Section 2.2.5 in the Python CSP framework. 116
A.6 Pseudocode of the continuous PSO with global best information [55] 117
A.7 A pseudocode of a discrete version [54] of the PSO in Figure A.6 118
A.8 Schoofs and Naudts' PSO for solving binary CSPs [90], named as *bcspPSO* in this research. 119
A.9 Distribute 5 variables to 9 particles for performing DFS. 121
A.10 A particle performs depth-first search on variable set $\{var_4, var_5, var_1\}$ . . . 122

A.11 Different non-DFS variable values generate different assignments in a graph colouring. . . . 122
A.12 Particles perform depth-first search in the graph colouring problem. 123

C.1 The mean evaluation value of *pop_rate*: problem sets 25 from the Comparison phase. . . . 162
C.2 The mean evaluation value of PSO models from the Comparison phase. . . . 163
C.3 The mean run time of PSO models from the Comparison phase. 164
C.4 The mean run time of PSO algorithms using the distance function. 165
C.5 The mean run time of PSO algorithms using the distance function. 166
C.6 The mean run time of PSO algorithms using the conflict count function. . . . 167
C.7 The mean run time of PSO algorithms using the conflict count function. . . . 168
C.8 The mean number of consistency checks of PSO models from the Comparison phase. . . . 169
C.9 The mean number of consistency checks of PSO algorithms using the conflict count. . . . 170
C.10 The mean number of consistency checks of PSO algorithms using the conflict count. . . . 171
C.11 The mean number of consistency checks of PSO algorithms using the distance function. . . . 172
C.12 The mean number of consistency checks of PSO algorithms using the distance function. . . . 173
C.13 The mean run time of PSO populations: problem set 25 from the Comparison phase. . . . 174
C.14 The mean number of consistency checks of PSO populations: problem set 25 from the Comparison phase. . . . 175

Chapter 1

Introduction

1.1 Motivation

Constraint satisfaction problems (CSPs) are a natural abstraction for many computational problems, and thus have been a major research topic in AI for many years [108]. This abstraction naturally represents real-world problems. Many problems such as scheduling, resource allocation and planning have been described as CSPs, and many techniques have been developed to solve those problems. Still, much attention is needed in various aspects of CSP research such as the development of new algorithms. In this research, we develop and evaluate new algorithms to solve general n-ary CSPs.

Research in swarm intelligence started in the late 1980s and has been attractive to AI researchers because it is simple and robust and offers a new alternative to solve many practical problems [4, 107]. As the name suggests, swarm intelligence models swarms of insects and birds. Through communication, these swarms are able to adjust their behaviour and to achieve their common objectives. Researchers have used these ideas to solve optimization problems [25, 42, 71, 72, 7]. Two popular techniques of this paradigm are ant colony optimization (ACO) [22] and particle swarm optimization (PSO) [53]. ACO models ants and PSO models birds. Both swarm techniques have been applied to solve random binary constraint satisfaction problems [89, 90, 99], but not general n-ary CSPs. Although it is possible to convert n-ary constraints to equivalent binary constraints [80], depending on the nature of the constraints, an n-ary CSP can become more difficult to solve after the conversion [102]. Besides, the process of converting an n-ary CSP to its equivalent binary CSP can be complicated and, not all the conversions can be done properly and produce

semantically equivalent representation [47, 102]. In addition, n-ary constraints provide a natural formulation for modelling real-world problems [86]. Thus, we do not want to limit our development for solving only binary CSPs. If we want to try something new to solve general CSPs, PSO is such a technique with potential not only because of the previous research in solving optimization problems and binary CSPs but also because of its nature of having multiple "workers" who can work individually and collaboratively to achieve a goal.

1.2 Research Goal

The goal of our research is to create new and effective particle swarm algorithms for solving general n-ary CSPs. Researchers have applied PSOs to solve various optimization problems and random binary CSPs. In this research, we will study these traditional PSOs, understand the interplay between the PSOs and CSPs, make the connection between them, and propose new techniques to develop new particle swarm algorithms for solving n-ary integer CSPs. Through the experimental results, we will answer whether the new particle swarms can solve general n-ary integer CSPs more effectively than the traditional ones. If they can solve n-ary CSPs, we would like to find out how we may possibly enhance these algorithms in the future.

1.3 Thesis Overview

This research combines CSPs and particle swarm optimization techniques, so in chapters 2 and 3, we review the background of CSPs and PSO respectively. In Chapter 2, we explain basic terminology, give CSP examples, review existing CSP problem solving techniques and also introduce the Python CSP framework used for this research. In Chapter 3, we look at swarm intelligence, review traditional PSO algorithms, and discuss PSO problem solving and its applications. Also, we raise our research questions as the goal of this research in Section 3.6. In Chapter 4, we specifically describe how to apply PSO to solve CSPs. We first introduce the problem representation to link CSPs and PSO together, explain the particle swarm algorithms we developed for solving CSPs and then formulate test problems to evaluate the swarm algorithms. In Chapter 5, we illustrate our experiments, evaluate our algorithms and analyze the results. In Chapter 6, we conclude our findings and propose future directions for further research.

Chapter 2

Constraint Satisfaction Problems

Since constraint-based ideas were first applied to solve AI (artificial intelligence) problems in the 1960s and 70s, constraint representations have been considered a natural way to describe many real-world problems. For decades, **constraint satisfaction problems** (CSPs) [108] have been one of the core research problems in AI, and the Association for Computing Machinery (ACM) has recognized constraint programming as one of the strategic directions in computer science research [38].

Many problems have been described as CSPs such as temporal reasoning, scheduling, network routing, DNA sequencing, puzzle matching, resource allocation, floor plan design, circuit design, graph problems and other combinatorial problems [108, 57]. CSPs are in general NP-complete [46, 59] and solving CSPs is hence NP-hard. CSPs typically represent problems as a set of **variables**, **domains**, and **constraints**. A domain of a variable is the allowable values for that variable, and constraints restrict which domain values the variables may simultaneously be assigned. The goal of solving a CSP is to find one or more legal **assignments** that satisfy all the constraints.

This chapter is organized in four sections. In the first two sections, we will introduce the terminology and some examples that we use throughout the chapter. Then a review of the existing constraint problem techniques follows. A new CSP framework in Python [20] used for this research will be described in the last section of this chapter.

2.1 Definitions

Before giving examples of CSPs, some terminology is needed. A CSP is formally defined as a tuple of (V, D, C) namely variables, domains and constraints [108, 2].

Definition 2.1.1. V, D and C of a CSP are defined as follows:

- $V = \{ v_1, v_2, \ldots v_n\}$ is a finite set of n **variables**. Each variable is a 'place-holder' that is able to hold an assigned value [61].

- $D = \{ D_1, D_2, \ldots D_n\}$ is a finite set of **domains**. Each domain is finite.[1] For each i in $\{1, 2, \ldots n\}$, domain D_i represents the set of all possible values $\{val_{i1}, val_{i2}, \ldots val_{ib}\}$ that can be assigned to the respective variable v_i. A variable-value **assignment pair**, $<v_i, val_{ij}>$ is also called a **label**, which assigns domain value val_{ij} to variable v_i, where $val_{ij} \in D_i$. A **complete assignment** refers to n such labels ($<v_1, val_1>$, $<v_2, val_2>$, $\ldots <v_n, val_n>$), one per variable. A **partial assignment** is a subset of a complete assignment. The domain of a variable can be a finite set of integers, real numbers, Boolean values, or any objects. However, for this thesis, we restrict domains to be finite sets of integers.

  In later chapters, we use two terms to describe the structures of CSP domains: a consecutive domain and consistent domains.

  1. A **consecutive domain** is a variable domain in which all the elements can be listed as a sequence of consecutive integers; i.e. the elements of a domain are consecutive. For example, $D_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ is a consecutive domain, whereas $D_2 = \{1, 3, 5, 7, 9\}$ is not. This consecutiveness becomes slightly more complicated when we discuss binary encoded domains in Section 4.2.1.2.

  2. If all the domains of a problem are the same, the problem has **consistent domains** or the domains of the problem are consistent. For example, we may have CSP domains $D_1 = D_2 = D_3 = \{1, 2, 3, 4\}$; then, domains D_1, D_2 and D_3 are consistent. On the other hand, we may have CSP domains $D_4 = \{2, 4, 6, 8\}$, $D_5 = \{1, 2, \ldots, 400\}$, $D_6 = \{0, 1\}$ and $D_7 = \{1, 2, \ldots, 10\}$; then, domains D_4, D_5, D_6 and D_7 are not consistent.

[1]Some research considers infinite domains; we only take finite domains into consideration in this thesis.

- $C = \{C_1, C_2, \ldots, C_p\}$ is a set of constraints. For each k in $\{1, 2, \ldots p\}$, $C_k(v_1, v_2, \ldots v_m)$ is an m-ary constraint and m can be 1, 2, ... to the size of the problem. A constraint is a Boolean function on the variables that restricts what values those variables can take simultaneously. If an assignment $(val_1, val_2, \ldots val_m)$ causes $C_k(v_1, v_2, \ldots v_m)$ to return true, the constraint C_k is satisfied by the assignment; otherwise, it returns false. If all constraints $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ are satisfied by one assignment, the CSP is satisfied and this assignment is **consistent**. In practice, other than being a function, a constraint can also be an equation, a logical relation, a set of all legal tuples (i.e. a **good list**) or a set of all illegal tuples (a **bad list**) on the variables.

Definition 2.1.2. An n**-ary CSP** is a CSP that contains n variables and each of the constraints in the problem may involve any number of variables between one and the size of the problem n.

Definition 2.1.3. $S = \{S_1, S_2, \ldots S_m\}$ is a set of all **solutions**. Generally, a CSP can have 0, 1, or more solutions. A solution $S_i \in S$ is a complete assignment $\{<v_1, val_{1a}>, <v_2, val_{2b}>, \ldots <v_n, val_{nm}>\}$ of all n variables in the CSP, where all the constraints are satisfied. A CSP is **solvable** if $|S| \geq 1$.

Besides finding perfect solutions as the above, a *good enough* solution to a CSP where only most of the (critical) constraints are satisfied, or an optimal solution to a CSOP (constraint satisfaction optimization problem) may also be possible depending on the nature of the problems [2].

Definition 2.1.4. A **penalty function** is a function that takes an assignment as its input and returns zero for a satisfiable assignment,[2] or returns a value greater than zero to penalize an unsatisfiable assignment.

It is a common technique in CSP research to evaluate the quality of an assignment. The smaller the penalty, the better the quality. Counting the number of constraint violations and estimating the distance from a potential solution to a satisfiable solution are some examples [90, 63]. Michalewicz et al. indicate that adaptively combining the maximum **completion cost**[3] with the expected completion cost can render a better penalty function [63].

---

[2] A satisfiable assignment is a potential solution that is feasible and does not violate any constraints.

[3] A completion cost is the cost to complete or obtain a satisfiable solution from a given assignment.

2.2 Examples

A CSP can be as simple as a one-variable arithmetic problem or as complex as a university scheduling problem with thousands of variables. What CSPs are and how a problem can be modelled as a CSP will become clear by the examples below. One should however know that the representations of a problem may not be unique, and different representations may affect the efficiency of finding solutions [27].

2.2.1 A warm-up example

Suppose we want to find a number between 1 and 10 that is an even integer. One obvious way to represent this as a CSP is:

- Variables: v
- Domains: $D_v = \{1, 2, \ldots 10\}$
- Constraints:
  - `even`(v) returns true if $val \in D_v$ is even for an assignment $v = val$.
- There are 5 solutions: $v = 2$, $v = 4$, $v = 6$, $v = 8$ and $v = 10$.

2.2.2 Pythagorean triple example

If we want to find the integer lengths of a right triangle where $\angle C = 90°$ and $0 < a, b, c \leq 30$ are the respective integer lengths of the three sides $\overline{BC}$, $\overline{AC}$ and $\overline{AB}$ shown in Figure 2.1:

- Variables: a, b and c
- Domains: $D_a = D_b = D_c = \{1, 2, \ldots 30\}$
- Constraints:
  - `pythagorean`(a, b, c) returns true if $a = val_a$, $b = val_b$ and $c = val_c$ satisfy $a^2 + b^2 = c^2$ relation where $val_a \in D_a$, $val_b \in D_b$ and $val_c \in D_c$.
- There are 22 solutions and one of them is $a = 3$, $b = 4$ and $c = 5$.

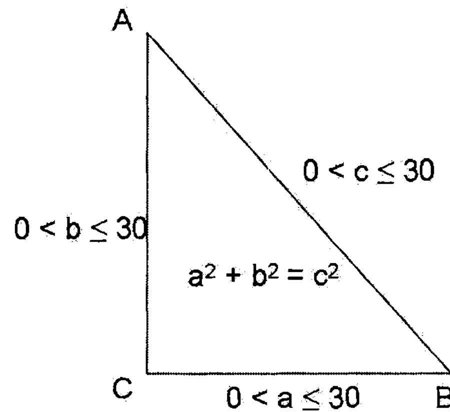
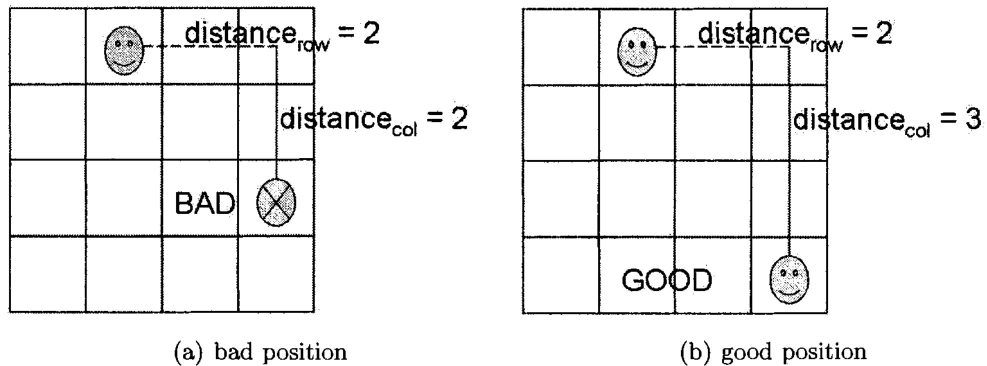


Figure 2.1: A Pythagorean triple example in Section 2.2.2.

2.2.3 8-queens problem

In the 8-queens problem, the problem is to place 8 queens on an 8-by-8 chessboard such that no two queens attack each other; in other words, no two queens can be placed on the same row, column or diagonal. One possible representation of this problem is to have 8 variables for the queens' row positions and one queen per column. To avoid queens being placed on the same row, the values of the variables must be all different. For every queen-pair, the row (Manhattan) distance should not be equal to the column distance so that the queens will not sit on the same diagonal as shown in Figure 2.2.

- Variables: q_1, q_2 $\ldots q_8$ for the positions to place queens on column 1, 2, …and 8 of the chessboard respectively.
- Domains: for each $i \in \{1, 2, \ldots 8\}$, domain $D_{qi} = \{1, 2, \ldots 8\}$ for variable q_i is a set of the possible row positions to place a queen on column i.
- Constraints:
  - `all_diff`(q_1, q_2, $\ldots q_8$) returns true iff the values of q_1, q_2, $\ldots q_8$ are pairwise different.
  - for every pair of queens (q_i, q_j), `undiagonal`(q_i, q_j) returns true only if $val_i \in D_{qi}$ and $val_j \in D_{qj}$ such that $|q_i - q_j| \neq |i - j|$ holds.



(a) bad position (b) good position

Figure 2.2: Queen positions in a 4-queens problem.

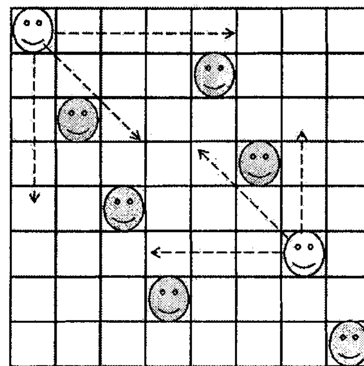


Figure 2.3: A sample solution of the 8-queens problem.

- Figure 2.3 shows one of the 92 solutions, where $q_1 = 1$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 2$, $q_6 = 4$, $q_7 = 6$ and $q_8 = 8$.

2.2.4 send-more-money Puzzle

The send-more-money problem shown in Figure 2.4 is an example of **cryptarithmetic puzzles.**[4] In this problem, we want to find a unique digit (0–9) for each letter s, e, n, d, m, o, r, y and satisfy the equation SEND + MORE = MONEY with no leading zeros (i.e. s

[4]There is a lot of information on the web, and http://www.clps.de/html/protcl/protcl/node62.html is one of them.

and m cannot be zero).

```
   s e n d                    9 5 6 7
 + m o r e    [solution]=>  + 1 0 8 5
 ---------                  ---------
 m o n e y                  1 0 6 5 2
```

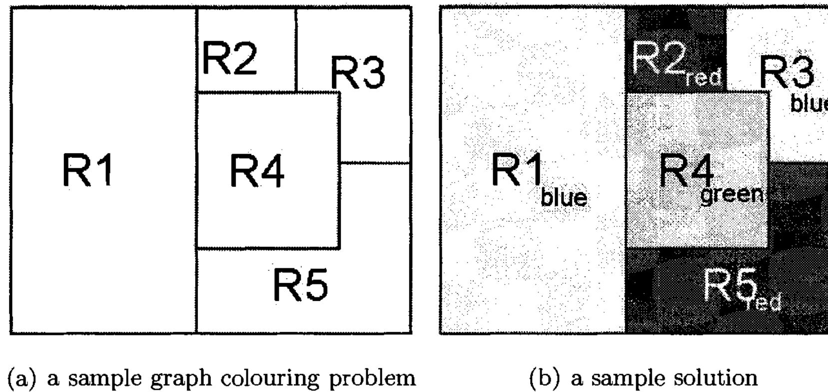
Figure 2.4: A send-more-money puzzle and its solution.

- Variables: s, e, n, d, m, o, r and y
- Domains: $D_s = D_m = \{1, 2, \ldots 9\}$ and $D_e = D_n = D_d = D_o = D_r = D_y = \{0, 1, 2, \ldots 9\}$
- Constraints:
  - `all_diff`(s, e, n, d, m, o, r, y) returns true only if the values of s, e, n, d, m, o, r, y are pairwise different.
  - An equation $(s \times 1000 + e \times 100 + n \times 10 + d) + (m \times 1000 + o \times 100 + r \times 10 + e) == m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y$ must hold.
- The unique solution of this problem is $s = 9$, $e = 5$, $n = 6$, $d = 7$, $m = 1$, $o = 0$, $r = 8$ and $y = 5$.

2.2.5 Graph colouring

The problem in graph colouring is to assign one colour for each region on a map from a selection of colours and the adjacent regions cannot be in the same colour. Assume we have red, green and blue to colour the map in Figure 2.5(a).

- Variables: r_1, r_2, r_3, r_4, and r_5
- Domains: $D_{ri} = \{\text{red, green, blue}\}$ for $i = 1, 2, 3, 4, 5$
- Constraints:
  - For every pair (r_i, r_j), $r_i \neq r_j$ must hold if r_i and r_j are adjacent.
- There are 6 solutions in the problem of Figure 2.5(a) and Figure 2.5(b) shows one, where r_1 = blue, r_2 = red, r_3 = blue, r_4 = green and r_5 = red.



(a) a sample graph colouring problem (b) a sample solution

Figure 2.5: A sample graph colouring problem with 3 colours.

2.3 Problem Solving Techniques

Many techniques to solve CSPs have been developed; some originate from solving other types of problems and some are specifically for solving CSPs. Basic CSP solving techniques include: search algorithms, problem reduction, and (hybrid) heuristic strategies. Some well-known techniques are described below.

2.3.1 Basic search algorithms

Search is a fundamental technique in AI problem solving.[5] The basic search algorithms mentioned here are simple and serve as a core to other sophisticated search methods. In this section, we will follow the convention to divide search algorithms into systematic and stochastic search. Generate-and-test [2, 57] and simple backtracking [36] are the examples of systematic search. Random guessing and random-walk algorithms are two stochastic search.

2.3.1.1 Basic systematic algorithms

2.3.1.1.1 Generate-and-test (GT) assigns values to variables and obtains a complete assignment; and then, it checks whether the assignment satisfies all the constraints. This

[5] http://pages.cpsc.ucalgary.ca/~denzinge/projects/know-search.html.

brute-force method tries all values to all variables one by one and checks for the consistency of the current assignment. Because no value is **pruned** or removed from the domain during the consistency checks, the complexity of the algorithm in the worst case is $|D_1 \times D_2 \times \cdots \times D_n| = O(d^n)$. Since GT eventually checks all possible combinations, it is a **complete** algorithm and all solutions can be found (given enough time) if there is any solutions. Because of the completeness, it can also be used to prove that a CSP is **unsatisfiable**, i.e. no solution exists.

2.3.1.1.2 Chronological backtracking (CBT or BT) systematically traverses the entire search space in a depth-first manner. It instantiates one variable at a time until it either finds a solution or runs out of instances and proves no solutions exist. Smarter than GT, the algorithm stops and backtracks as soon as it finds the partial assignment inconsistent. More specifically, if the instantiated variables so far do not violate any constraint, the algorithm keeps going on to the next variable; otherwise, it backtracks to the previously assigned variable and reassigns an untried value. Backtracking is **sound** and complete,[6] but it can be inefficient because of **thrashing**. It does not identify the actual culprit of the inconsistency so it may keep failing and backtracking for the same reasons again and again [57].

Chronological backtracking can be effective for simple problems; what is more important, the strategy is so useful that most **systematic** search algorithms (as opposed to **stochastic** methods) extend or derive from it to improve the performance.

2.3.1.2 Basic stochastic algorithms

2.3.1.2.1 Random guessing algorithm is the most naive stochastic search. Like blindly throwing darts, it repeatedly 'guesses' a complete assignment and checks if the assignment satisfies the constraints until it finds a solution or reaches timeout (or some maximum number of iterations). Since the algorithm assigns variables in a non-systematic way, it neither avoids checking for the same assignment repeatedly, nor guarantees to verify all possible assignments. Because it does not guarantee to check all possible assignments, the algorithm is **incomplete** and so it cannot guarantee a solution or prove no solutions.

On the other hand, the method is so simple and fast that it can be used for a problem with many solutions if any solution is acceptable. Also, the randomness can be used to

[6]It guarantees that the solution returned must be correct. Also, the algorithm guarantees to find a solution if there exits a solution.

quickly estimate the solution density of a search space.

2.3.1.2.2 Random-walk algorithm (RW) is a basic local search technique [83]. It initializes all the variables and "walks" through the search space fixing inconsistencies one at a time until it satisfies all the constraints or times out. Similar to random guessing, it is incomplete and so cannot guarantee a solution or prove no solutions. To prevent the walk from wondering too much, many heuristics have been developed. In addition, random-walk can serve as a simple strategy for other stochastic search methods to escape from local optima, which will be discussed in Section 2.3.4.

2.3.2 Problem reduction techniques

Searching for solutions can be very time consuming, especially if the search space is big and the solutions are distributed in a haphazard way. To improve the efficiency, one can sometimes trim the size of the search space and simplify the original problems. Problem reduction [108] is such a method that can be used at the beginning of a search or during a search. Once a problem becomes smaller and simpler, search algorithms can go through the space faster. In some cases, problem reduction can solve CSPs without searching [108]. However, some reduction techniques may be too complex and too expensive in practice. Often, problem reductions are applied to supplement other search strategies.

In this section, we will review several common problem reduction techniques such as *node-consistency* and *arc-consistency*.[7] Generally, these techniques are derived from the idea of a **constraint graph** [108, 2], where the nodes represent the variables of a CSP and the edges are the constraints indicating the relationship among the variables. Any two or more connected edges become a path. Figure 2.6 illustrates the constraint graph of the graph colouring problem in Section 2.2.5.

Each of these problem reductions provides a different level of consistency; for instance, node-consistency guarantees the consistency in the nodes (i.e. the individual variables). Depending on the problems and the level of consistency required, some combination of these techniques can be used.

---

[7]Refer to [108] for other problem reduction techniques such as *path-consistency* and *k-consistency*.

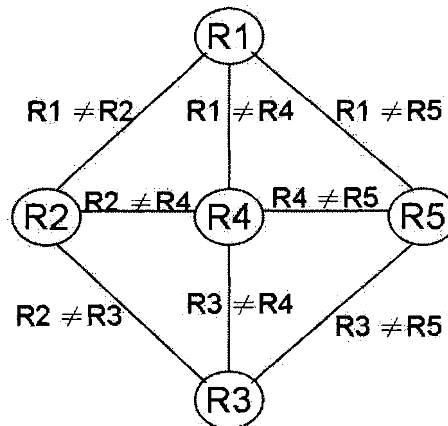


Figure 2.6: A constraint graph of the graph colouring problem in Section 2.2.5.

2.3.2.1 Node-consistency and arc-consistency

The simplest problem reduction techniques are node-consistency (NC) and arc-consistency (AC), which prune the inconsistent values from the variable domains. The difference between these two is that node-consistency affects individual variables (or unary constraints) whereas arc-consistency checks binary constraints between two variables. NC ensures consistency at the node level and AC ensures it on the edges on a constraint graph.

All AC algorithms check binary constraints, but in various ways. Some commonly used ACs are noted as AC-1, AC-2, ...AC-7. The higher number generally indicates that the AC provides a higher degree of consistency, but at greater cost. For the effectiveness and efficiency of the problem solving, AC-3 and AC-4 [98] are the two most widely used. Another AC variant, *directional arc-consistency* (DAC), performs consistency checks on directed edges. In providing consistency level, DAC is not so strong as the ACs above but can be quite efficient [108].

2.3.3 Strategic search and heuristics

Because neither basic search nor consistency checks alone can always solve CSPs in a timely manner, adding heuristics and using hybrid algorithms are often used to improve performance. This is not only applicable to the systematic search, but also true of the stochastic search described in Section 2.3.4.

2.3.3.1 Constraint propagations

Problem reduction techniques in Section 2.3.2 can also be applied to interleave between search steps to continuously narrow the search space, which is called *constraint propagations* [2]. Backmarking, backchecking, backjumping and forward checking are some examples of this sort [30, 31, 37]. Forward checking (FC) for example, looks ahead and performs consistency checks while assigning a value to a variable and removes inconsistent values from the unassigned (future) variables. If any future variable has no consistent values available, the algorithm backtracks as in chronological backtracking. By looking ahead to remove impossible assignments, FC is an efficient general-purpose search algorithm [83].

2.3.3.2 Hybrid strategies

Some algorithms explore multiple branches at the same time [108] and other techniques mix the strength of various algorithms and integrate different search strategies. To list a few, here are some examples that combine forward search with backtrack search: backtracking with backjumping (BMJ), backtracking with conflict-directed backjumping (BM-CBJ), FC-BJ, FC-CBJ, MAC-BJ and MAC-CBJ [73, 74].

2.3.3.3 Variable-orderings and value-orderings

Another common strategy is to vary the order in which variables and domain values are searched. Generally speaking, search orderings shape the structure of a search space, which often affect the search results and efficiency. By ordering the variables differently, different search spaces are constructed without changing the complexity of a problem [108]. Imagining doing a tree search, variable orderings define the orders of the branches. Variables may come in different domain sizes and thus turn out different **branching factors**. Once the search order has been changed, the order of branching factors can be varied and the effective search space may become bigger or smaller to the algorithms. Many strategies utilize such characteristics and determine different orderings. For example, applying Haralick and Elliott's **fail-first** principle [37], a search algorithm can dynamically pick the variable that either has the smallest (remaining) domain or is involved in most constraints. Dealing with such variables first, the algorithm is able to detect dead-ends early.

Similarly, value orderings change the search structure as well. Different from variable orderings however, value orderings rearrange nodes within a branch and try to find a node

(or a value) that is most likely to succeed. One popular value ordering is the min-conflict heuristic [64]. The idea is to select a value that can either minimize the total number of conflicts or minimize the conflicts with other unassigned variables. Although the min-conflict can be used with a complete search and becomes for example an informed backtracking [65], it is more commonly used with stochastic search methods and more detail will be discussed in the following section.

2.3.4 Stochastic search

Complete systematic methods are often inefficient for large hard CSPs [112]. Fast and good enough solutions are sometimes desirable and acceptable in many real world applications. In contrast to systematic search, stochastic methods wander in the problem space in a relatively nondeterministic manner. Local search methods have been popular for quite a long time and evolutionary algorithms have drawn the researchers' attention in the recent years [112].

2.3.4.1 Local search

Local search is a **repair based** strategy, where a fully assigned initial assignment is generated first (often randomly, but not necessarily) and then the assignment is repaired or improved until a solution is found or some timeout mechanism kicks in. Hill-climbing is a typical example, which improves the search results iteratively based on an evaluation function. At each time step, one of the best **neighbours** will be chosen to be the next assigned state. In CSPs for example, the evaluation function can aim at minimizing the number of constraint violations and the (best) *goal* is to have 'zero' violation. A general hill-climbingrandomly selects the *next* state from the candidate neighbours if there is more than one candidate with the same evaluation. In other hill-climbing based algorithms, heuristics such as the min-conflict, are used as a guidance to the neighbour selection process.

Generally, local search methods are incomplete and cannot guarantee to find a solution or to prove no solutions. One reason is that they have no idea which assignment node they have or have not visited, and another is that they may get stuck in a local optimum or wander around on a plateau. Much research has been done in preventing, detecting or escaping from these situations. For instance, random-walk [92] uses noise to walk out of traps. Tabu search [34, 101] avoids going into the same bad area again by remembering

its previous experience. Simulated annealing [1, 70, 18] models the cooling speed of the annealing procedure to improve the performance.

2.3.5 Evolutionary Computing

Evolutionary computing (EC) includes genetic algorithms (GAs) [40, 35, 19], evolutionary programming, evolutionary strategies (ES) and genetic programming [55]. It has become a popular problem solving paradigm in AI. Swarm intelligence (SI) originated from a cellular robotic system [6] is closely related to evolutionary computation methods. These EC techniques were originally developed for optimizing numerical functions, training neural networks and so on. They have been applied to CSPs, for example in [60, 8, 14, 89, 88, 99, 90]. These problem solving techniques generally use a population of possible solutions, fitness information and probabilities in tackling problems [55]. Often, they first generate the initial states of the population, and iteratively alternate between evaluating the fitness value of the candidates and evolving the new states of the population until certain stopping criteria arrive.

2.3.5.1 Genetic algorithms

Simulating biological genetic systems, the development of genetic algorithms (GAs) started in the 1950s [55]. These algorithms have been applied to solve some constraint satisfaction optimization problems (CSOPs) and CSPs [109, 60, 8, 14] since the 90s. Based on the evolutionary theories, the idea is that the better fit individuals have a better chance to survive, and gradually the fitness of the population evolves and reaches its optimum. Specialized encodings, fitness functions or operators such as Michalewicz's GENOCOP [78] are often found in genetic algorithm research for solving problems [63, 14]. The choices of the parameter settings can sometimes depend on the problems and are one of the difficulties for users to apply.[8]

2.3.5.2 Swarm intelligence

Swarm modelling is inspired by the analogy of social insects, birds, fish and human cognition. It is a new optimization technique that emerged in the early 1990s. Chapter 3 will discuss

[8] Website http://w3.ualg.pt/~flobo/psgea-2005/ for "Workshop on parameter setting in genetic and evolutionary algorithms, PSGEA 2005"

particle swarm optimization in detail.

2.3.6 Summary of algorithms

For a quick review of the algorithms mentioned in this chapter, we summarize them in Table 2.1. These algorithms are characterized in terms of the completeness, the consistency, whether they make initial complete assignment or not, and if they are able to give a *potential* solution (not necessarily to be a consistent solution) at anytime while searching.

Table 2.1: A summary of algorithms

Algorithm	Type	Complete	Consistent	Initial	Anytime solution
generate-and-test	systematic	yes	yes	no	no
backtracking	systematic	yes	yes	no	no
forward-checking	systematic	yes	yes	no	no
random guessing	stochastic	no	yes	yes	yes
random-walk	stochastic	no	yes	yes	yes
hill-climbing	stochastic	no	yes	yes	yes
min-conflict HC	stochastic	no	yes	yes	yes
genetic algorithm	stochastic	no	yes	yes	yes
particle swarm	stochastic	no	yes	yes	yes
ant colony	stochastic	no	yes	yes	yes

2.4 CSP Frameworks

Besides the traditional research in search algorithms and problem solving techniques, CSP research also includes the development of libraries and frameworks. According to Roy et al. [81], "a framework approach integrates objects with constraints to provide extensible and flexible implementations". BackTalk [82], ILOG Solver [45] and JCL (the Java Constraints Library) [58] are some examples. BackTalk allows defining complex finite domain CSPs. It can be used as either a library for users to apply for solving CSPs or a framework for developers to design and implement their own CSP algorithms [81, 82]. ILOG Solver is a constraint-based optimization engine that provides optimization technology for scheduling, sequencing, timetabling, configuration, dispatching and resource-allocation applications with logical constraints [75, 45]. JCL is a Java based library for constraint satisfaction problems and it is able to handle both discrete finite domains and continuous domains [58].

2.4.1 Arithmetic CSPs in Python

We have developed a CSP framework in Python for solving **arithmetic** CSPs where constraints are modelled in arithmetic functions [20]. Python supports object-oriented programming and includes many interesting features such as operator overloading, generators, and list comprehension. With these features and the declarative nature of Python, we can make the CSP framework easy to understand and use. So far, this framework supports finite integer domains and contains a general backtracking solver. In this research, I will enhance this framework by implementing and comparing several different particle swarm optimization algorithms and some modified backtracking algorithms to solve (arithmetic) CSPs with finite integer domains.

2.4.1.1 Examples

By using the examples described in Section 2.2, we can illustrate how the CSP framework works. Taking advantage of Python language, formulating a CSP is very straightforward. What the framework users need to do is to define the variables (and domains), create the problem (CSP) and add the constraints one by one. If the users would like to solve the problem, they can specify a solver to solve the problem. Besides the CSP examples shown in Appendix A (Figure A.1 $\sim$ Figure A.5), a complete sample execution in Python is given in Figure 2.7.

```
>>> # import the csp framework
>>> from csp import *
>>>
>>> # define variables:
>>> # enumerate color red = 1, green = 2, blue = 3
>>> # range(1,4) render a list [1,2,3]
>>> R1, R2, R3 = var(range(1,4)), var(range(1,4)), var(range(1,4))
>>> R4, R5 = var(range(1,4)), var(range(1,4))
>>>
>>> # create a CSP:
>>> csp = problem(R1, R2, R3, R4, R5)
>>>
>>> # add constraints:
>>> csp += R1 != R2
>>> csp += R1 != R4
>>> csp += R1 != R5
>>> csp += R2 != R3
>>> csp += R2 != R4
>>> csp += R3 != R4
>>> csp += R3 != R5
>>> csp += R4 != R5
>>>
>>> # solve the problem:
>>> for sol in gen_backtracking(csp):
        print 'R1 = %s, R2 = %s, R3 = %s, R4 = %s, R5 = %s'\
            % value(R1, R2, R3, R4, R5)


R1 = 1, R2 = 2, R3 = 1, R4 = 3, R5 = 2
R1 = 1, R2 = 3, R3 = 1, R4 = 2, R5 = 3
R1 = 2, R2 = 1, R3 = 2, R4 = 3, R5 = 1
R1 = 2, R2 = 3, R3 = 2, R4 = 1, R5 = 3
R1 = 3, R2 = 1, R3 = 3, R4 = 2, R5 = 1
R1 = 3, R2 = 2, R3 = 3, R4 = 1, R5 = 2
>>>
```

Figure 2.7: A Sample Run of the graph colouring problem in Python CSP framework.

gen_backtracking(csp) is a function call to a Python backtracking generator, which gives solutions one at a time when it is called until it runs out all solutions.

Chapter 3

Particle Swarm Optimization

3.1 Introduction

Particle swarm optimization (PSO) [53] is a popular problem solving technique in the swarm intelligence (SI) paradigm. It was first introduced by Kennedy and Eberhart in 1995. They developed simple methods which could efficiently optimize continuous nonlinear mathematical functions. Borrowing ideas from artificial life (A-life), social psychology and swarming theory [53, 55], PSO simulates swarms such as flocks of birds and schools of fish searching for food.[1]

Also, PSO is related to evolutionary computation (EC), but it is somewhat different [44, 53]. Similar to many EC techniques, PSO initializes a problem state to a population of randomly distributed solutions. Unlike many other ECs however, PSO "evolves" solutions based on individual experience and group experience, rather than using evolutionary operators (e.g. the crossover and the mutation operators in genetic algorithms). It assumes that socially shared information helps its population evolve. In other words, the population iteratively updates and searches for optima with the shared information.

Since its first development, many PSO variants have been evolved, and research has shown promising results in some well-known test functions [53, 96, 10, 55]. Starting with some SI techniques in Section 3.2 and several PSOs in Section 3.3, this research will investigate the PSO techniques specifically for solving the constraint satisfaction problems discussed in Chapter 2.

[1] Craig Reynolds' *boids* [77] is a well-known visual simulation of flocking.

3.2 Swarm Intelligence

A swarm is "a collection of organisms or agents which interact with one another [25]." The term *swarm intelligence* (SI) was first used in reference to Beni and Wang's cellular robotic systems in the late 1980s [6, 3]. In their systems, a group of simple robots interact with their neighbour robots via communication. Later (in the early 90s), swarm intelligence studies were inspired by social insects, birds, fish and human cognition. In the recent years, swarm modelling has become a new strategy for solving both constrained and unconstrained optimization problems [42, 71, 72, 7]. In addition to solving optimization problems, limited research has also applied to solve CSPs [89, 99, 90]. Among others, ant colony optimization (ACO) [22] and particle swarm optimization [53] are two popular swarm systems. PSO will be investigated starting from Section 3.3. In this section, we will briefly look at the ACO techniques.

3.2.1 Ant Colony Optimization

Marco Dorigo invented ant colony optimization by modelling insect ants' behaviour to solve discrete optimization problems, such as the travelling salesman problem [22, 21]. Artificial ants have been tested on other combinatorial problems such as quadratic assignment problems, graph colouring, job-shop scheduling, sequential ordering, network routing, swarm-based robotic problems [6, 25] and constraint satisfaction problems [89, 99].

The idea is that a colony of ants collaboratively find shortest paths between their nest and a food source without a coordinator or leadership [25]. These ants "communicate" through a chemical substance called *pheromone*. In ACO, this pheromone represents the information shared among the fellow ants. Essentially, each individual ant leaves pheromone on the way it goes by and picks up pheromone left by the other ants. Because this substance gets weaker over a distance and gradually evaporates over time, the amount of pheromone implies how frequently and recently ants have been through the same path. Stronger pheromone may indicate that a path is shorter or the chances of being successful is higher.

In the algorithm, ants randomly wander the search space at first. Once pheromone accumulates on several trails, individual ants detect it and choose a path among the candidate trails. This decision making will be favorable to those with stronger pheromone deposit. Eventually, these ants will converge to an optimized path and find a solution.

3.2.1.1 Ant colonies for solving CSPs

Because ACO is closely related to PSO and it has been shown to solve constraint satisfaction problems particularly on binary CSPs, we would like to briefly examine these systems. Schoofs and Naudts' systems [89] and Solnon's solvers [99] show that ACO can perform better than several other evolutionary algorithms (EAs) and competitively with forward-checking conflict-directed backjumping (FC-CBJ) on a set of random binary CSPs [113].

3.2.1.1.1 Schoofs and Naudts' ant systems. Schoofs and Naudts propose two ant systems to solve binary CSPs [89, 88]. One is based on a constraint graph and uses a standard penalty function[2] counting for constraint violations and evaluating the quality of a potential solution. Another system uses a hybrid penalty function and path consistency. Making use of path consistency, their hybrid system is capable of not only reducing the size of the search space, but also showing the insolvability outside of a **mushy region.**[3] One problem as indicated by the authors is that even with this hybrid system, they still could not solve a problem or prove the problem unsolvable in the mushy region.

3.2.1.1.2 Solnon's ant solvers. Solnon presented three ant solvers [99]. Those ant colonies have been shown to solve random binary CSPs rather effectively and efficiently compared with several well-known EAs, a random-walk and FC-CBJ [99, 113]. Although some parameter settings may influence the performance, they are more algorithm specific. Two strategies are worth noting: adding local search and preprocessing assignments. In the first case, the algorithm performs local search to locally improve the quality of a complete assignment as soon as such an assignment has arrived. In the latter case, it initializes a set of complete assignments called **SampleSet** with some local search preprocessing; and then, the solver starts with these initialized SampleSet and searches for solutions. The author intentionally keeps the ant solvers as generic as possible, but she comments that this may indeed have the solvers handle some global constraints less efficiently than specialized algorithms [99].

[2] See Definition 2.1.4 in Section 2.1.

[3] A mushy region is also referred to as a *phase transition* region, where a substance is neither entirely in one phase nor in another (e.g. liquid vs. solid). The random CSPs in such a region with different tightness of constraints, contain both solvable and unsolvable instances [97].

3.3 Traditional PSO

Kennedy and Eberhart developed the first particle swarm algorithm in 1995 to simulate how the birds fly synchronously [53, 23, 25]. This simulation then became known as the particle swarm optimization search algorithm. PSO's origin and the relations to other scientific research have been discussed in Section 3.1. In this section and the next section, we will first define the common terminology and introduce the original PSO algorithms; and then, we will review some PSO techniques. In Section 3.5, we will investigate a PSO [90] which solves binary constraint satisfaction problems. In the last section, we will state the research questions of this thesis.

3.3.1 Definitions

In PSO, a problem is modelled as an n-dimensional solution space and a population of particles search through this n-dimensional space for optimal solutions.

Definition 3.3.1. In PSO, a **particle** P_i simulates an individual in a bird flock. Figure 3.1 shows a group of particles in a 2-dimensional space. Each particle in the group is responsible for searching and keeping solutions together with its fellow particles. At any time t, particle P_i is located at some **position** $x_i(t)$ in the n-dimensional problem space. Conventionally, $x_i(t)$ indicates the current position of P_i and $x_i(t-1)$ represents the previous position. In the problem solving context, a particle with its position represents a potential solution.

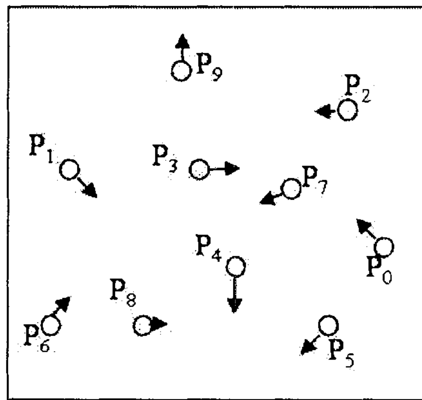


Figure 3.1: Particle swarm (population = 10) in a 2-dimensional space.

Definition 3.3.2. In PSO, a **swarm** $P = \{P_1, P_2, ...P_s\}$ is a set of particles.

Definition 3.3.3. A particle's **velocity** $\vec{v_i}(t) = [u_1, u_2 \ldots, u_n]$ is an n-dimensional vector that moves particle P_i at time t as shown in Figure 3.2. Mathematically, the position-velocity relation is

$$x_i(t) = x_i(t-1) + \vec{v_i}(t) \tag{3.1}$$

In PSO, velocities are mainly affected by particle's own knowledge and the neighbours' experience. Conceptually, a velocity $\vec{v_i}(t)$ can be derived from the relation in Equation 3.2, where φ_1 and φ_2 are parameters as will be discussed in Section 3.3.2.1. According to this relation, a velocity can be computed using Equation 3.3.

$$\vec{v_i}(t) = \varphi_1(individualexperience) + \varphi_2(globalexperience) \tag{3.2}$$

$$\vec{v_i}(t) = \omega\vec{v_i}(t-1) + \varphi_1(experience_i) + \varphi_2(experience_g) \tag{3.3}$$

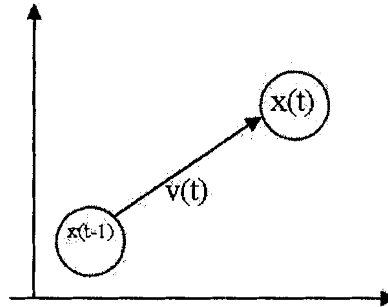


Figure 3.2: A position-velocity relation in a 2-dimensional space.

Definition 3.3.4. A **neighbourhood** defines the social structure of a swarm and indicates which particles a particle should interact with. Within a neighbourhood, particles interact, communicate and share information. To form a neighbourhood, we may not restrict to the physical distances between particles; in fact, they are often defined by the enumeration labels of the particles in PSO [25]. For example in Figure 3.3, nine particles are enumerated as P_1, P_2, $\ldots P_9$. Regardless of the physical distance, P_1, P_2 and P_3 are **neighbours** of size three,[4] and P_4, P_5 and P_6 form another. Stars, rings and wheels are the most commonly

[4]According to the research papers in the field, size 3 neighbourhood is sometimes denoted as $k = 2$

used neighbourhood structures (shown in Figure 3.4).

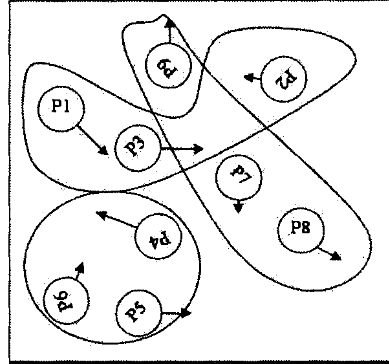


Figure 3.3: A global swarm vs. local neighbourhoods [25].

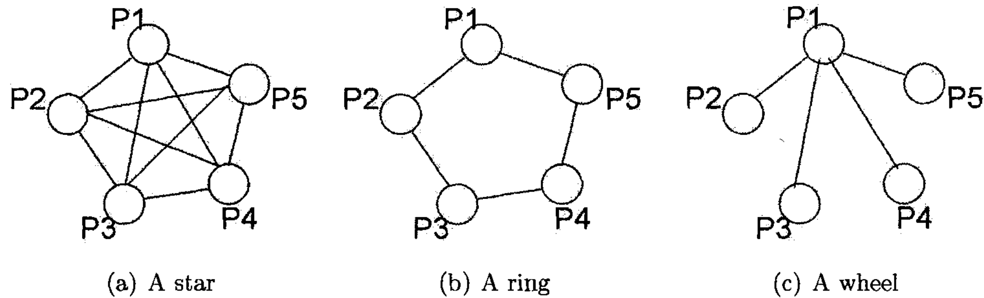


Figure 3.4: Simple neighbourhood topologies (population = 5) [51, 25].

In the PSO context, two terms, **local** versus **global** are often used. "Local" refers to an individual neighbourhood while the global refers to the entire swarm as one big neighbourhood. For example, there are three local neighbourhoods in Figure 3.3. Neighbourhoods can overlap and a particle can belong to multiple neighbourhoods. For instance, particles P_1, P_2, P_3, P_4 and P_5 are to form neighbourhoods of size 3 in a ring topology as shown in Figure 3.4(b). We may have five neighbourhoods in total: $\{P_1, P_2, P_3\}$, $\{P_2, P_3, P_4\}$, $\{P_3, P_4, P_5\}$, $\{P_4, P_5, P_1\}$ and $\{P_5, P_1, P_2\}$. A particle in such a structure retrieves information from another two particles directly connected to it.

Different neighbourhood structures may affect the performance of the swarm. They determine how information propagate among particles, and thus may affect the **convergence** of particles, i.e. when and how particles may come together, arrive at some stable state and stop improving the solution. That is, particles may converge on different local optima or at different time with different neighbourhood **topologies**.[5] In a star topology as shown in Figure 3.4(a), all particles are influenced by one *global best* location so far in every iteration and move towards the location, so they tend to converge quickly to the *global best*. In a ring topology, the neighbourhood segments are overlapped so the convergence may spread from one neighbourhood to another and eventually pull all the particles together. By gradually spreading information, the swarm converges slower in a ring than in a star. For a swarm in a wheel, there exists one and only one central particle, which serves as a *buffer* [55]. The central particle collects and compares the positions of all particles, finds the best one and moves itself towards the best position. All other particles then pull information from the central particle and start moving towards the same position. Because of this buffering effect, a wheel topology may preserve diversity for a bit longer and prevent the swarm from converging too fast on local optima.

PSO evaluates the quality of its solutions based on an **objective function**, **fitness function** or **evaluation function** $F(x)$. By evaluating and comparing the current solution with the best solution found so far, a particle determines its next move. Three *best* solutions (or positions) so far are the individual best (*pbest*), global best (*gbest*) and local best (*lbest*).

- The individual best position $xpbest_i$ refers to P_i's best position found so far, $xpbest_{ij}$ is the jth element of $xpbest_i$, and the individual best $pbest_i$ is the evaluation on $xpbest_i$.
- The global best position so far $xgbest$ is the best position so far of the swarm P and the global best $gbest$ is the evaluation of $xgbest$. Similarly, $xgbest_j$ is the jth element of $xgbest$. Although this can be used by particles in other neighbourhood structures, it is typically used in a star topology where all particles exchange information in a single neighbourhood [25].

[5] A topology refers to the logical structure of a swarm neighbourhood, rather than a physical structure. We have mentioned, neighbouring relations are often defined by the enumeration labels of the particles in PSO. For example in Figure 3.3, nine particles are enumerated as P_1, P_2, $\ldots P_9$. Regardless of the physical distance, P_1, P_2 and P_3 are neighbours of size three. P_4, P_5 and P_6 form another one, and P_7, P_8 and P_9 are the other.

- The local best position so far $xlbest_k$ is the best position so far of all particles in neighbourhood k and the local best $lbest_k$ is the evaluated value. A ring topology is an example where particles use this information [25]. Particles in a ring topology often forms a number of (local) neighbourhoods; in each neighbourhood, particles share a local best position. Note that global best *gbest* is a typical example of *lbest* where all particles of a swarm form a single (global) neighbourhood.

3.3.2 Continuous PSO

PSO was originally designed to optimize continuous nonlinear mathematical functions, and so it deals with real numbers [53]. The algorithm randomly initializes each particle P_i to position $x_i(0)$ and velocity $\vec{v_i}(0)$. At each time step t, every particle calculates a new velocity $\vec{v_i}(t)$ based on the *social-psychological* tendency [25, 53] from both its own and its neighbours' knowledge. Considering different ways of sharing information, there can be three ways to compute velocities:[6]

- Individual *pbest* only or one particle per neighbourhood: each particle makes decisions on its own, and ignores everybody else.

$$\vec{v_i}(t) = \omega \vec{v_i}(t-1) + r_1 c_1 (xpbest_i - x_i(t-1)) \tag{3.4}$$

- Global *gbest* and individual *pbest*: every particle considers the knowledge of all particles within a single neighbourhood.

$$\vec{v_i}(t) = \omega \vec{v_i}(t-1) + r_1 c_1 (xpbest_i - x_i(t-1)) + r_2 c_2 (xgbest - x_i(t-1)) \tag{3.5}$$

- Local neighbourhoods *lbest* and particle individual *pbest*: suppose particle P_i is in neighbourhood k

$$\vec{v_i}(t) = \omega \vec{v_i}(t-1) + r_1 c_1 (xpbest_i - x_i(t-1)) + r_2 c_2 (xlbest_k - x_i(t-1)) \tag{3.6}$$

Once the new velocity has been determined, particle P_i updates its position using Equation 3.1 mentioned earlier. Then iteratively, all particles keep updating the velocities and their positions until timeout or the goal fitness value is obtained.

[6] ω is a parameter to control how much the new velocity is affected by the previous velocity. r_1 and r_2 are random numbers in [0, 1] to randomize the influence of group experience and particles' individual experience. c_1 and c_2 are positive **acceleration constants**.

In short, this algorithm makes use of a swarm of particles stochastically and intelligently exploring new regions and exploiting towards the previous *better* regions until the swarm reaches an "optimum". The particles' intelligence comes from social interaction and information sharing, and such learning abilities dominate the PSO algorithm [55].

3.3.2.1 Parameters and variants

Like many other evolutionary algorithms, the behaviours of the individuals and the population (the particles and the swarm in PSO) are affected by the parameters in the algorithm; and so tuning the parameters changes the performance of the search [115, 25]. Considering the algorithm and Equation 3.5, a generic PSO has a list of parameters to work with:

- A particle position and a velocity vector are a node and a vector in an n-dimensional solution space. Both of them consist of n elements. The dimension n is also the size of a problem to solve, and so the solution space is n-dimensional. For example, the size of a constraint satisfaction problem (CSP) is the number of variables in the problem.

- The population *pop* of the swarm depends on the problem to solve. Between 10 and 50 are commonly used *pop* values [23] and some experiments use 100 as their *pop* values.

- A size k neighbourhood consists of $k+1$ particles. There is no standard neighbourhood size. Research indicates 15 percent of the swarm size as neighbourhood size can be useful [23].

- **Inertia weight** ω determines the influence of the previous velocity $\vec{v_i}(t-1)$ [55, 95] and in turn controls particle's ability to explore and exploit the space. It also affects the speed of particles converging or **de-converging**, i.e. the speed of pulling the particles together or preventing the particles from settling. To get a better searching pattern between global exploration and local exploitation, researchers recommended to decrease ω over time from 0.9 to 0.4 [95, 55, 23]. By doing so, the particles can explore widely at the beginning and gradually shift to exploit towards an optimum.

- The relative magnitudes between $\varphi_1 = r_1 \times c_1$ and $\varphi_2 = r_2 \times c_2$ upper bounds determine whether a particle move towards the neighbour best (*gbest*) or the individual best (*pbest*). If we have the upper bound of φ_1 greater than the upper bound of φ_2, particles tend to emphasize neighbours' experience and move towards the neighbour

best (*gbest*). r_1 and r_2 are random numbers between 0 and 1, which bring randomness to φ_1 and φ_2 and affect the **acceleration constants** c_1 and c_2. These random numbers are set at each calculation of a velocity so that particles may vary the influence between different sources of information. Unlike r_1 and r_2, the acceleration constants c_1 and c_2 are controllable. If they are small, the velocity gradually becomes smaller so the particle tends to slow down over time, and vice versa.

- Velocities are obtained stochastically because of the randomness introduced by r_1 and r_2. So, the upper bound of velocities V_{max} is needed to prevent particles from exploring here and there forever and not being able to converge. If V_{max} is too big, particles may fly too far at once and miss good solutions. If V_{max} is too small, particles may be limited to a local area. But, the choice of V_{max} is problem dependent, for instance, the size of a variable domain in CSP context. Based on the researchers' experience [23, 55], V_{max} can be set at 10 to 20% of the range of each variable or proportion to the range of the problem. For example, suppose the search range of a variable is in [100, 200]. V_{max} can be set to between 10 and 20.

- A **constriction coefficient** χ was added by Clerc to Equation 3.5 [10, 12, 55].

$$\vec{v_i}(t) = \chi(\vec{v_i}(t-1) + r_1c_1(xpbest_i - x_i(t-1)) + r_2c_2(xgbest - x_i(t-1))), \quad (3.7)$$
$$\text{where } \chi = \frac{2k}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}, k \in]0,1[\text{ and } \varphi = r_1c_1 + r_2c_2 > 4.0$$

When φ increases, χ becomes smaller and so particles take smaller steps through the search. In turn, the magnitude of the coefficient affects the convergence of the swarm. Researchers suggest that a simple setting $\chi \approx 0.729$ where $k = 1.0$ and $\varphi = 4.1$ seems to work well [52]. The mathematical analysis of the coefficient is not in the scope of this research. Refer to [12] for more detail.

Some researchers show that when using χ or ω to control velocities, the upper bound V_{max} is unnecessary [12]. Some others indicate that one may still have a better control on particles to explore or exploit with V_{max} although it is not required [24, 55]. For example, studies show that having V_{max} set to the maximum potential solution X_{max} to Clerc's model, the elements of a velocity are individually controlled by the upper bound and the swarm does not seem to get stuck so easily at a local optima [55].

3.3.3 Discrete (Binary) PSO

Kennedy and Eberhart's discrete model [54] is a version of the PSO that does not directly use real numbers. It makes the PSO applicable to problems with variable values taken from a discrete domain e.g. $v \in \{1, 1.5, 2, 2.5, 3\}$ as opposed to over a continuous range $1 \leq v \leq 3$ where there are infinite number of values between any two numbers. The rationale is that not all problems can be described using continuous domains; for example, the graph colouring in Section 2.2.5 has finite domains such as {red, blue, green}.

In Kennedy and Eberhart's discrete PSO, a particle and its position still represent a solution in the problem solution space. Instead of consisting of a sequence of integers or real numbers however, a particle P_i's position $x_i(t)$ at time t is composed of a bit-string: $x_{i1}(t), x_{i2}(t) \ldots, x_{in}(t)$ where $x_{ij}(t) \in \{0,1\}$ for each $j \in \{1, 2, \ldots n\}$. Also, in order to derive the bit value of $x_{ij}(t)$, a velocity element $v_{ij}(t)$ is not directly used as an increment to compute $x_{ij}(t)$, rather it is used as a threshold to determine the possibility of a bit change. More specifically, $v_{ij}(t)$ is transformed by a *sigmoid* function and then compared with a uniformly distributed random number $\rho_{ij}(t) \in [0,\ 1]$.

$$x_{ij}(t+1) = \begin{cases} 0 & \text{if } \rho_{ij}(t) \geq \frac{1}{1+exp(-v_{ij}(t))}, \\ 1 & \text{otherwise.} \end{cases} \tag{3.8}$$

3.3.3.1 Binary encodings

In order to apply the binary representation to integer domains or real numbers, an integer or a real number must be converted to a bit-string. Two common encoding methods are Binary encoding and Gray encoding [55]. Binary encoding uses regular binary numbers (in base 2) to represent integers or real numbers; for example, '001', '010' and '011' represent integer 1, 2 and 3. Gray encoding also converts integers to sequences of bits. The only difference is Gray encoding minimizes the bit changes between consecutive numbers; for example the Gray encodings of 1, 2 and 3 are '001', '011' and '010' so the bit change between 1 and 2 is one bit rather than 2 bits in Binary encoding. Since Gray encoding flips only one bit at a time when the corresponding number increments one, Kennedy et al. recommend Gray encoding. It is suggested that Binary encoding may introduce undesired complexities to problem solving because the Hamming distance between any consecutive numbers is not uniform and it can be harder to systematically control the changes from 0 ('000') to 1 ('001') or from 1 ('001') to 2 ('010') as needed for example [55].

3.4 Solving Problems with PSOs

3.4.1 Research problems and applications

PSO was originally developed as an optimization problem solver. It is most commonly applied to optimization problems or to those problems that can be converted to one [23]. Finding minima or maxima of a nonlinear function is a typical test problem for PSO algorithms [25, 96]. Also, it has been shown to optimize global unconstrained optimization problems [72, 71]. Some well-known static, numerical, continuous, real-valued constrained benchmark problems are popular among PSO researchers as well [115].

The early PSO applications were related to artificial neural networks (NNs) [55, 25]. Because of the success in training neural networks, PSO has been applied to various related applications such as human tremor analysis and diagnosis, Parkinson's disease prediction, rule extraction and computer controlled milling optimization [55, 25, 96]. In addition to the neural networks, several constrained nonlinear optimization problems have been investigated. Studied applications are, for example, neural network training [10, 85], human tremor analysis and diagnosis [93], ingredient mix optimization problem [55], computer controlled milling optimization [106], reactive power and voltage control [116], power supply reliability enhancement [68], internal combustion engine design [76], and so on. More and more variants are being developed.[7] Several possible application areas suggested by Eberhart and Shi [23] are multi-objective optimization, pattern recognition, scheduling, and so on. Refer to [23] for more suggestions and examples.

3.4.2 Strengths

One reason for PSO gaining its popularity is that it is conceptually straightforward and computationally simple [55]. Simulating birds flocking, particle swarms fundamentally use two simple formulae to effectively search the goal. Also, research has shown that in comparing PSO with other algorithms on a variety of problems [15, 87, 5, 33, 118], it can perform better on some problems and be competitive on others. Since PSOs are a new search technique, much research has been targeting to improve the original PSOs for solving various problems and it has great potential to be done further. For example, owing to its similarity

[7] For most recent research, one can refer to *Particle Swarm Optimization* website [41], which keeps a list of bibliography and related information.

to evolutionary computation (EC) methods, many successful EC techniques and ideas may be integrated to improve PSOs. Like many EC algorithms, PSO has a number of parameters to adjust. On one hand, this is beneficial for implementing adaptive systems [55] and also shows the extensibility of PSO to other specifically designed algorithms although it may not perform as well as those algorithms. On the other hand, tuning parameters for solving a particular problem or a range of problems can be time-consuming and non-trivial. Compared with EC methods, PSO does not have as many parameters to tune in order to get acceptable performance [42]. In addition, Hu and Eberthart suggest that PSO is applicable for both constrained and unconstrained problems even without pre-transforming the constraints and the objectives of a problem [42].

3.4.3 Weaknesses

Researchers have found several issues that prevent the generic PSOs from effectively solving certain types of problems. Although the improvement has been working on to handle these issues, the solutions may not easily be applied to solve other problems; thus, we should keep these issues in mind while developing new particle swarms for solving other problems. For example, although PSO has the ability to converge quickly, it tends to wander and slow down as it approaches an optimum [115]. Owing to the premature convergence, it gets stuck quite easily and cannot explore wide enough. This can be problematic for solving **multimodal** problems where the problems have multiple optimal solutions. Particularly if many of those optima are only local rather than global [115], particles may get trapped at local optima. In addition, while there are not many parameters to control [42] and as mentioned previously, these parameters open up a potential for developing adaptive PSO systems, some of the parameters are problem dependent. Some suggested values and experimental settings are still at trial-and-error stage [23], and it can be non-trivial to find the right settings for individual problems.

3.5 Solving binary CSPs

Although quite a number of PSO variations are designed to solve constrained and unconstrained optimization problems, the first and the only one developed specifically for solving CSPs was done by Schoofs and Naudts in 2002 [90]. Similar to their research in ant systems [89], this PSO solves binary CSPs and was tested on a set of randomly generated binary

CSPs. Besides a *no-hope/re-hope* mechanism for preventing particles from getting stuck in local optima, several additional operators were introduced to calculate particles' positions and velocities. Schoofs and Naudts conclude that the system is able to solve the tested random binary CSPs reasonably well but not so good as an ant colony algorithm and a genetic algorithm on hard problems [90].

3.5.1 Schoofs and Naudts' operators

The spirit of the original PSO and the meanings of the original formulae (Equation 3.1 and Equation 3.5 for computing a particle's position and velocity) remain essentially the same. Mathematically however, those two formulae have been reformulated with the new operators as follows to compute a velocity and update a position [90]:

$$\vec{v}(t) = (\varphi_1 \otimes (\vec{xpbest} \ominus \vec{x}(t-1))) \circ (\varphi_2 \otimes (\vec{xgbest} \ominus \vec{x}(t-1))) \quad (3.9)$$

$$\vec{x}(t) = \vec{x}(t-1) \oplus \vec{v}(t) \quad (3.10)$$

where velocity $\vec{v}(t)$ is a vector of $[v_1, v_2, \ldots, v_n]$ at time t, a particle position $\vec{x}(t)$ consists of $[x_1, x_2, \ldots, x_n]$ at time t, $\vec{xpbest}$ and $\vec{xgbest}$ denote the individual best position so far and the global best position so far respectively, and parameters φ_1 and φ_2 will be explained in the next section. These elements are either the same or similar to those in the original PSO. The difference is how they are computed via those new operators. Much detail of these operators are described in [90]. Briefly, they are

1. $\ominus$ denotes a position change from one to another. $(\vec{x} \ominus \vec{y} = \vec{v})$ moves from position $\vec{x}$ to $\vec{y}$ and results in a velocity $\vec{v}$ where $\vec{v}$ is a vector $[y_i \rightarrow x_i]$.

2. $\oplus$ is a reverse operator of $\ominus$; it calculates the next position after a position change with a velocity vector. Computationally, $(\vec{x} \oplus \vec{v} = \vec{y})$ adds a velocity $\vec{v}$ to a position $\vec{x}$ and becomes a new position $\vec{y}$.

3. $\circ$ operator adds two velocities and yields a new velocity. Specifically, suppose we have $\vec{v} \circ \vec{w} = \vec{u}$ where $\vec{v} = \vec{b} \ominus \vec{a}$ and $\vec{w} = \vec{y} \ominus \vec{x}$; then for each element u_i in $\vec{u}$, u_i is either $a_i \rightarrow y_i$ if $b_i = x_i$, or otherwise $a_i \rightarrow b_i$.

4. $\otimes$ is used to multiply a velocity $\vec{v}$ with a coefficient φ and to render a new velocity $\vec{w} = \varphi \otimes \vec{v}$. Representing CSP in PSO terms,[8] suppose a variable x_i is an element of

[8] Refer to Section 4.1 for details.

position $\vec{x}$ and its **conflict counts** ($nbConf_i$: the number of constraint violations the current value of variable x_i causes) is greater than some given φ, we can obtain an element w_i with respect to $\vec{w} = \varphi \otimes \vec{v}$ by having $x_i \rightarrow x_i$, or otherwise $w_i = x_i \rightarrow y_i$ if $nbConf_i$ is not greater than φ. This should become clear in Chapter 4.

3.5.2 Parameters used in Schoofs and Naudts' PSO

The parameters used Schoofs and Naudts' PSO are:

- Like other PSOs, Schoofs and Naudts' particle position and velocity vector are a node and a vector in an n-dimensional solution space. They consist of n elements. The dimension n is also the size of a constraint satisfaction problem (CSP), i.e. the number of variables in the problem.

- The population *pop* is the size of the swarm. The value to use depends on the problem to solve.

- Coefficients φ_1 and φ_2. Schoofs and Naudts examined both $\varphi_1 = \varphi_2 = 0$ and $\varphi_i = \varphi_2 = 1$, and the experiments showed that the solution quality is better when $\varphi_1 = \varphi_2 = 0$ [90].

- A *deflection* operator gives the probability of direction changes of a particle to refine the particle's moving direction. Schoofs and Naudts compare this operator to the mutation operators in genetic algorithms. Refer to [90] and the algorithm in Figure A.8 in Appendix A for the usage of the operator. This operator was set to 0, $1/n$ or $2/n$ in [90]. According to Schoofs and Naudts, the deflection feature seems to reduce the instability of the velocities caused by the no-hope mechanism so the level diversity can be maintained at a certain level. But, if the deflection is set too high, the swarm will move too fast to focus.

3.6 The Research Goal: PSOs for Solving n-ary CSPs

we learned that particle swarm optimization (PSO) has been successfully applied to various constrained optimization problems, and also proposed as a technique for solving random binary CSPs [90]. In our research, we will extend the ideas of the traditional PSOs and

Schoofs and Naudts' PSO, and develop new particle swarms to solve general n-ary integer CSPs.[9] For the purpose of this research, we will want to answer the following questions:

- Schoofs and Naudts developed a PSO algorithm for solving binary CSPs [90]. They modified the traditional PSOs by a set of new operators and mathematically reformulated the computation of the velocity and position of the particles with the new operators. They tested their algorithm on random binary CSPs and reported good results on non-hard problems. Can we use their algorithm or extend the algorithm to solve general n-ary integer CSPs effectively?
- The traditional Continuous PSO [53] and Discrete PSO [54] were not designed for CSPs. The questions are:
    1. How can we modify these traditional PSOs to solve n-ary integer CSPs?
    2. How do the algorithms extending the traditional PSOs compare with Schoofs and Naudts' algorithm?

As we have discussed in Section 3.4.2 and Section 3.4.3, the traditional PSOs have their strength and a number of weaknesses. While designing and developing the new particle swarm algorithms for solving n-ary CSPs, we should make use of their strength and pay special attention to the weaknesses since CSPs are hard multimodal problems in general.

[9]See Section 2.1 for the definition as needed.

Chapter 4

Particle Swarm Optimization for Solving CSPs

To achieve our research goal and answer the research questions presented in Section 3.6, we will begin with proposing and developing particle swarm algorithms for solving n-ary constraint satisfaction problems (CSPs). Specifically in this chapter, we explain how we model CSPs in particle swarm, how we modify and improve the three particle swarm optimization (PSO) algorithms in Chapter 3 to search finite integer space, and then how we formulate a PC configuration problem as a CSP for the experiment. The experimental results will then be discussed in Chapter 5.

4.1 CSP Representation in Particle Swarm

In order to use PSO for solving CSPs, we must first put PSO to CSP solving and represent CSPs in a form that is searchable for PSO. Suppose a CSP has n variables var_1, var_2, ... var_n, and each variable can be assigned a single value from a finite set of integers (i.e. a CSP domain). Each of these integer sets is finite, but not necessarily consecutive over a range. The constraints imposed on the variables may be unary, binary, ternary or even more. In general, they may involve any number of variables between 1 and n.

4.1.1 Connecting CSPs and PSO

To solve CSPs using PSO, we need to determine the search space of the swarm, the particles, and the position and velocity of each particle. Firstly, PSO conventionally models a size n problem as an n-dimensional search space and the size of a CSP is measured by the number of variables, so we use the number of CSP variables to define the search space dimensions, one CSP variable per dimension. In any dimension i, the possible values to search are the domain of variable var_i. One search node in the PSO's search space is a complete assignment $\langle val_1, val_2, \ldots, val_n \rangle$ to the CSP variables var_1, var_2, ..., var_n respectively.

As a member of the swarm, a particle P_i takes part in the mission to search CSP solutions. Its position $x_i(t)$ as shown in Figure 4.1 represents a *potential* solution or CSP assignment found at time t. An element $x_{ij}(t)$ in a position $x_i(t)$ denotes a value val_j selected from CSP domain D_j. Similarly, particle P_i's best position so far $xpbest_i$, the swarm's best position so far $xgbest$ or the kth neighbourhood best position so far $xlbest_k$ are the best (potential) solutions so far found by P_i, by the entire swarm or by the kth neighbourhood, respectively.[1] They are all sequences of CSP assignments $\langle val_1, val_2, \ldots, val_n \rangle$.

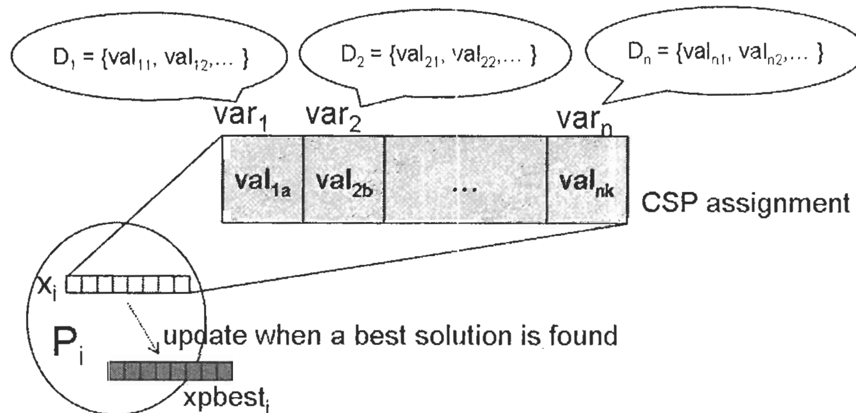


Figure 4.1: A particle position x_i in the CSP context is a complete assignment.

A velocity $v_i(t)$ of a particle P_i is an n-dimensional vector that moves P_i from its previous

[1] To determine its velocity to move from a current position to another, a particle takes two pieces of information into account: the individual best experience *xpbest* and the group experience. The group experience *xgbest* or *xlbest* depends on either the global or the local neighbourhood structure of the swarm discussed in Section 3.3.1.

position $x_i(t-1)$ to the next position $x_i(t)$. In the context of CSP, this velocity updates the complete assignment from one to another and each element $v_{ij}(t)$ of $v_i(t)$ in fact changes the assignment of variable var_j from one domain value to another. In Figure 4.2 for example, $v_{ij}(t) = 4$ increments domain value assignment $x_{ij}(t-1) = 2$ by 4 and yields $x_{ij}(t) = 6$.

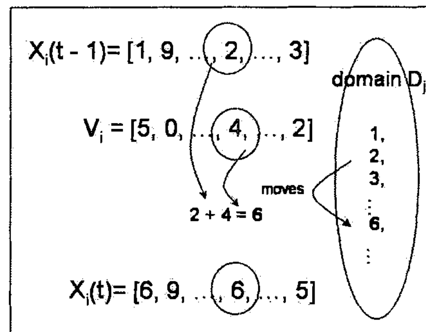


Figure 4.2: A particle velocity v_i changes CSP variable assignment.

4.1.2 Handling constraints

Moving from one position to another, a particle may violate constraints. Thus, what we need to resolve next is how PSO handles constraints. PSO was originally designed for optimization problems and it relied on an evaluation function $eval(x)$ to guide the particles. Intuitively, we can use such a mechanism to handle constraints in CSPs. Our task is to choose a good function, which can closely estimate the quality of an assignment and lead the swarm to a good solution quickly. For example, we can use a penalty function discussed in Section 2.1. In this research, we use two general penalty functions: a conflict count function and a distance estimation function. The conflict count function checks constraint violations and returns the arity of a constraint as a penalty score when the constraint is violated, or returns zero otherwise. The distance estimation function computes the distance from a potential solution to a satisfiable solution. For example, an assignment $a = 100$ violates a constraint '$a+5 \leq 50$'. In order to satisfy the constraint, a can only be at most 45. The distance estimation function returns (100 – 45 = 55).

Besides employing penalty functions to guide the particles, we consider several other constraint handling strategies such as repairing **infeasible solutions** or unsatisfiable potential solutions to improve the search or to minimize the constraint violations [63, 13]. The

details are to be discussed in Section 4.2.2.

4.2 PSO Algorithms for Constraint Satisfaction

As part of this research, several new particle swarm algorithms were developed based on the three PSO models discussed previously: the Continuous PSO (Section 3.3.2), the Discrete PSO (Section 3.3.3) and Schoofs and Naudts' PSO for solving CSPs (Section 3.5). We will refer to them as the Continuous model,[2] Discrete model and BCSP model from now on.

One of the major challenges in applying PSOs directly to finite integer CSPs is that all assignments are only integers within a possibly non-consecutive range. This increases the complexity of the problems because the particles no longer *fly* smoothly in a continuous space. Instead, they need to *hop* in the space like a frog. In the first part of this section, we will discuss how to modify the three PSO models to search integer domains. Then, we will describe strategies that we propose to improve the algorithms.

4.2.1 Generic PSO

To keep the merits of PSOs, we want the algorithms to be as simple and as close to the originals as possible. The first modification makes PSOs work with finite integer CSPs; particularly, the Continuous model and Discrete model were not originally designed to deal with integer CSPs. This modification allows each element $x_{ij}(t)$ of a particle position $x_i(t)$ take on a value only from its corresponding CSP (integer) domain D_j. The BCSP model works with finite integer domains so no such modification is necessary. We will refer to these PSOs with minimal changes as generic type PSOs of all three models.

4.2.1.1 Continuous model: *genericPSO*

As mentioned before, we modified the continuous PSO to work with finite integer domains. The algorithm allows each element of a particle position[3] to take on values only from its integer domain. Specifically, we modify the algorithm in updating velocities and positions as shown in Figure 4.3, Line 4–7. The computation of the velocity and position of a particle

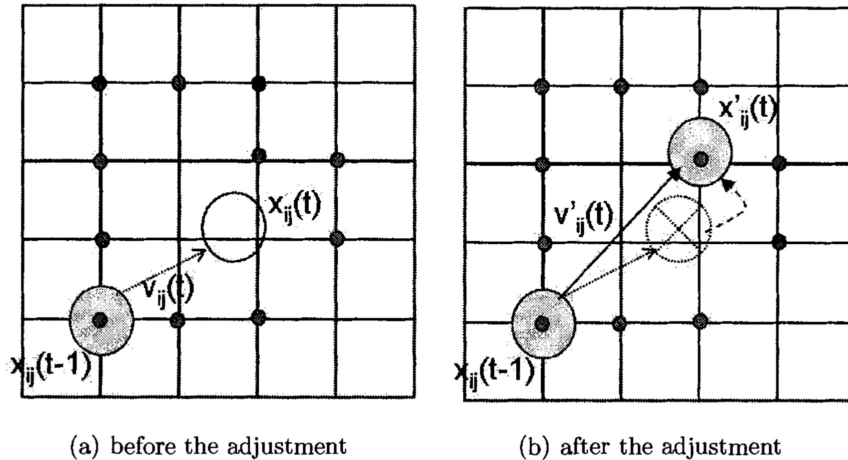
[2] Although we will modify the model to handle discrete integer CSP domain, we still refer to it as Continuous model.

[3] i.e. $x_{ij}(t)$ in $x_i(t) = \langle x_{i1}(t), x_{i2}(t), \ldots, x_{in}(t)\rangle$ for $j = 1, 2, \ldots, n$

P_i is done one element (dimension) at a time, for n dimensions. Figure 4.4 illustrates the idea where $v_{ij}(t)$ moves an element $x_{ij}(t-1)$ of a position to $x_{ij}(t)$. If the resulting $x_{ij}(t)$ is in domain D_j, no change is necessary; otherwise, we adjust $v_{ij}(t)$ to $v'_{ij}(t)$ and $x_{ij}(t)$ to $x'_{ij}(t)$ so $x'_{ij}(t)$ can be in D_j and close to $x_{ij}(t)$.[4]

```
1  FOR j = 1, 2, ..., n:
2     v[j] = update velocity
3     x[j] = update position x[j] + v[j]
4     IF x[j] not in domain D[j]:
5        x[j] = relocate to x'[j], where
6               x'[j] is the closest location to x[j] in D[j]
7        v[j] = adjust to v'[j] so that x[j] = x[j] + v[j] maintains
```

Figure 4.3: A pseudocode segment describes the change done for the Continuous PSO.



(a) before the adjustment (b) after the adjustment

Figure 4.4: A position adjustment on the jth-axis when the element x_{ij} goes out of domain D_j.

Assuming particle P_i is at $x_i(t-1)$. The small dots represent the values in domain D_j, and the filled circle $x_{ij}(t-1)$ is the jth element of $x_i(t-1)$. Suppose $v_{ij}(t)$ does not move $x_{ij}(t-1)$ to another small dot on the board, as shown on the left. The algorithm will adjust $v_{ij}(t)$ to $v'_{ij}(t)$, find the closest "small dot" and move the particle to $x'_{ij}(t)$.

[4]In this research, particles only search within CSP domains due to the limitation of the CSP Framework. Alternatively, one may consider keeping $x_{ij}(t)$ as long as it is an integer; otherwise, $x_{ij}(t)$ can be moved to an integer $x'_{ij}(t)$ close to $x_{ij}(t)$. If $x_{ij}(t)$ in the former case or $x'_{ij}(t)$ in the latter case is not in D_j, one should count it as a constraint violation.

4.2.1.2 Discrete model: *binaryDiscrete* and *grayDiscrete*

We also apply the same idea described in Section 4.2.1.1 to the Discrete model. The difference is that the Discrete PSO [54] represents a potential solution in an m-bit string[5] instead of a sequence of n integers. Unless the CSP domains are binary, some additional modification should be done beforehand. One possible change is to directly encode each integer element to its corresponding bit string. If each integer is encoded as an x-bit string, the original n-dimensional particle position becomes an xn-bit string. For example, we may have a continuous *genericPSO* particle at location $\langle 6, 4, 1, 9\rangle$. Transformed by binary encoding [55], this position in *binaryDiscrete* is $\langle 0110010000011001\rangle$.[6] Or with a Gray encoding [55] in *grayDiscrete*, the position becomes $\langle 0101011000011101\rangle$.[7] Our preliminary experiment has shown several immediate disadvantages of these representations to the Python CSP framework introduced in Section 2.4.1.

The first issue is the speed. The discrete solvers *binaryDiscrete* and *grayDiscrete* tend to be slower than the continuous *genericPSO* over the same number of iterations. For a problem of size n, assuming each integer domain value can be encoded to an x-bit string, a binary encoded position of a particle becomes xn dimensions whereas an integer encoded position is n dimensions. In other words, at each iteration each particle of the discrete solvers computes velocities and updates particle positions for $x \times n$ times, but the particle of the continuous solver performs this computation only for n times.

Another issue is the "non-consecutive" CSP domains.[8] Since the integers of a CSP domain may not be consecutive, not all bit strings correspond to legal CSP assignments. Even with a consecutive domain, binary bit flops may produce some undesirable bit strings as explained below.

1. Inconsistency among CSP domain sets may cause the undesirable bit strings.[9] For consistent implementation, we set all the bit string segments[10] at the same length, regardless of what variable domain a bit string comes from. The string segments of

[5] m-bit string consists of $x_{i1}x_{i2}\ldots x_{im}$, for example, $011\ldots0$.

[6] To present it clearly, we may divide the 16-bit binary encoding into 4 segments $\langle 0110, 0100, 0001, 1001\rangle$, one segment per original integer value.

[7] i.e. $\langle 6, 4, 1, 9\rangle = \langle 0101, 0110, 0001, 1101\rangle$ in Gray encodings

[8] Domain consecutiveness has been defined in Definition 2.1.1.

[9] Domain consistency has been defined in Definition 2.1.1.

[10] one segment for one original integer value of a particle position.

$\langle 0001, 0011, 0101, 0111 \rangle$, for instance, are all length 4 bit segments, and the segments of $\langle 000000110, 110010000 \rangle$ are length of 9. Technically, we take the maximum domain value of all variable domains to determine the length of bit segments. However, this representation may waste space and time if the ranges of all domains are different. For example, we may have CSP domains $D_1 = \{2, 4, 6, 8\}$, $D_2 = \{1, 2, \ldots, 400\}$, $D_3 = \{0, 1\}$ and $D_4 = \{1, 2, \ldots, 10\}$. To have the same length for each bit segment, we encode a 9-bit string for each domain value.[11] In this example, we obviously waste 19 additional bits (6 for D_1, 8 for D_2 and 5 for D_4) and these 19 bits consume unnecessary computation.

2. Another situation occurs within a CSP domain. For example, we have a domain $D = \{1, 3, 5, 7, 9\}$. We need 4 bits (from 0000 to 1111) to encode the domain and it becomes $\{0001, 0011, 0101, 0111, 1001\}$. This encoding appears fine at first. However when a particle computes velocities and performs bit changes, some undesirable numbers[12] may appear. These undesirable numbers directly affect the validity of the bit change and are not acceptable to the Python CSP framework.[13]

The situations discussed above may affect the effectiveness and the efficiency of the solvers. In order to compensate the problem, we have tried to adjust and reduce the probability of producing certain bit changes according to the distribution of CSP domains. But the adjustment may break the probability of a bit change from the original studies of the discrete PSO and lose the velocity information for the iterative particle movements. Our preliminary experiment has shown the disadvantage when CSP domains are not consecutive or not consistent.

---

[11] The maximum domain value "400" is encoded as 110010000, which is a length 9 bit string.

[12] These numbers are 0000, 0010, 0100, 0110, 1000, 1010, 1011, 1100, 1101, 1110 or 1111; these values do not exist in the variable domain.

[13] Another possibility is to enumerate a domain set and encode the values as a shorter and more compact representation such as $\{1, 3, 5, 7, 9\} = \{000, 001, 010, 011, 100\}$, to reduce the sparse effect. This may be possible if we have only a few CSP variable domains or each integer value across domains can be encoded into the same binary string. If we have a large number of variables and domains, the situation will become much more complicated because each domain set may contain different elements. So, we need to set up a lookup table for converting between the integer values and their binary strings for each domain. In addition, if the elements of CSP domains are encoded differently, we may have a problem in verifying constraints. For example, we may have two variables var_1 and var_2. The domains of var_1 and var_2 are $D_1 = \{1, 3, 5, 7, 9\}$ and $D_2 = \{9, 19, 29\}$ respectively. If we try the compact encoding, we will have $D_1 = \{000, 001, 010, 011, 100\}$ and $D_2 = \{000, 001, 010\}$. In such encodings, the value 9 in D_1 differs from the one in D_2. It may then require more careful thought when we examine a relation such as "9 == 9". One thing we know is that we cannot compare "100 == 000" directly.

4.2.1.3 BCSP model: bcspPSO

Schoofs and Naudts' PSO (short for BCSP-PSO) is designed for solving binary constraint satisfaction problems [90]. The particles of this algorithm update their positions within CSP variable domains, so they do not go out of domain. The first thing to consider is the *deflection* operator which is used as a Boolean predicate in the algorithm with an '`if` statement' in [90]. Functionally, the operator should give a probability to change the moving direction of a particle. Hence, we modify the statement to "`if random() < deflection`" as shown in Line 13 of Figure 4.5, where `random()` returns a random number between 0 and 1. With such a modification, the *deflection* becomes a probability threshold rather than a boolean switch. The value $1/n$ or $2/n$ for $deflection$ gives different probability of the direction change of a particle as discussed in [90], and the value 0 for $deflection$ sets the probability to zero.

```
1   randomly initialize the particles
2   set gbest, all lbest's and all pbest's to some very big values
3   initialize best positions xgbest, all xlbest's and all xpbest
4   t ← 1
5   while t < maximum number of iterations:
6         do for i ← 1 to population:
7               do for j ← 1 to n:
8                     do nbConf ← conflict counts of x_ij[t − 1] of particle P_i
9                        if nbConf > φ1:
10                          then v' ← xpbest_ij ⊖x_ij[t − 1]
11                          else  v' ← x_ij[t − 1] ⊖ x_ij[t − 1]
12                       if nbConf > φ2:
13                          then if random() < deflection:
                                    #comments: it was 'if deflection' in [90]#
14                                  then v'' ← Rand(j) ⊖ x_ij[t − 1]
15                                  else  v'' ← xgbest_j ⊖ x_ij[t − 1]
16                          else  v'' ← x_ij[t − 1] ⊖ x_ij[t − 1]
17                       x_ij[t] ← x_ij[t − 1] ⊕ (v' ∘ v'')

18                 fitness_i ← conflict counts in particle P_i
19                 if fitness_i < pbest_i:
20                    then xpbest_i ← x_i
21                         pbest_i ← fitness_i
22                 if pbest_i does not change for noHope times:
23                    then randomly initialize x_i

24          gbest, xgbest ← update from pbest, xpbest
25          lbest, xlbest ← update from pbest, xpbest
26          t ← t + 1
27  return gbest, xgbest
```

Figure 4.5: Schoofs and Naudts' PSO [90]. It is named as *bcspPSO* and serves as the foundation of all algorithms derived from BCSP model in this research.

4.2.2 Strategic PSOs

Our early experiments showed that the generic type PSOs[14] could not solve n-ary integer CSPs effectively. One possibility is that without enough diversity (or distance) among particles, the swarm converges and confines to a local optimum quickly. Another possibility is that particles tend to get stuck more easily with integer CSP search space. Particles' velocities are real numbers. In a continuous search space, particles can move in a much smaller scale (with real number velocities). In integer CSPs, the search space is discrete or even sparse because of the non-consecutive domains. If particles do not have enough power and the velocities are too small to get out of the current positions, the particles may keep going back to the same position. For example, suppose we have a domain $D_j = \{1, 2, 3, \ldots 10\}$ and a particle's position x_j is currently at 2. In a real number domain, the particle can possibly move gradually from 2.1, 2.2, ... to 3. However in an integer domain, the velocity must be big enough to take the particle to move from 2 to 3; otherwise, the particle has to stay at 2.

In this section, we will describe several modifications done to the generic type PSOs in order to avoid particles getting stuck and to enhance particles' exploration abilities. Also, efficiency is another issue that we will have to consider.

4.2.2.1 Zigzag movements: genericZigzag, binaryZigzag, grayZigzag and bcspZigzag

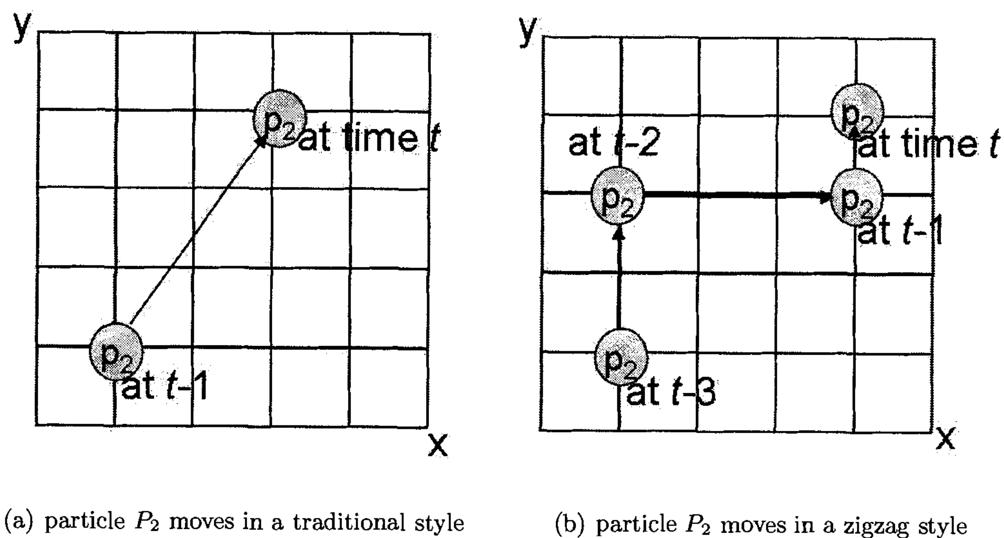
Regardless of the original PSO models, moving in all n dimensions at one time may have brought in too much driving force to the particles. This may have contributed to the fast convergence of the swarm because each particle is affected by the same best experience in all n dimensions at one time and tends to come quickly to the best experience so far. Also, such a big step prevents particles from examining a local area. For example, imagine we are in a 2 dimensional space, an x-y plane. Moving one step in both x and y with a velocity (2, 2) from location (0, 0), we can only visit one location at (2, 2). If we break the step into two smaller steps in x with velocity (2, 0) and then in y with velocity (0, 2), we may visit two locations (2, 0) and (2, 2). Besides, if step one and step two are affected by different best experience, we may visit (2, 0) and (2, 3) instead of (2, 0) and (2, 2) for instance.

Moreover in CSPs, a constraint violation can sometimes be fixed by changing the values

[14] with the basic modification in Section 4.2.1

of only one or two variables involved instead of changing the values of all variables. This is particularly true when a solution is close. Therefore, we propose to have particles move one dimension at a time in a "zigzag" manner as shown in Figure 4.6. Each of these zigzag type PSOs will be referred to as *genericZigzag* of the Continuous PSO, *binaryZigzag* and *grayZigzag* of the Discrete PSO with different encodings, and *bcspZigzag* of the BCSP PSO.

Computationally, this "zigzag" movement should increase the speed of each iteration because a particle only needs to compute one dimension (v_{ij}) of a velocity and update one element (x_{ij}) of a position each time instead of the entire n dimensions $v_{i1}, v_{i2}, \ldots, v_{in}$ and all n elements $x_{i1}, x_{i2}, \ldots, x_{in}$. This may also speed up the evaluation of a potential solution each time because these zigzag particles only need to examine one CSP variable in each iteration. Although this may possibly increase the total number of iterations the swarm requires to solve a problem, the swarm has potentially more chances to examine more solutions locally and to find a solution quickly. In addition, while moving one dimension at a time, each movement may be guided by a different best found so far; in turn, it may reduce the chance for the particles to be trapped in one particular best location so far.



(a) particle P_2 moves in a traditional style

(b) particle P_2 moves in a zigzag style

Figure 4.6: A particle moves in a 2-dimensional space with 2 different styles.

4.2.2.2 No-hope and re-hope: genericRestart

To escape from local optima, one **lazy** solution is to take actions only when the situation occurs. For example, the swarm may detect a **no-hope** situation when there has been no improvement made for a certain amount of time. As soon as the no-hope situation is detected, some **re-hope** action may take place to give the swarm more hope to continue finding solutions. This idea comes from Schoofs and Naudts' "no-hope & re-hope mechanism" [90] and has been built into the BCSP model including *bcspPSO* and *bcspZigzag*. Schoofs and Naudts' "no-hope & re-hope" mechanism simply restarts when a no-hope count[15] arrives. Schoofs and Naudts suggest that the swarm gets stuck quickly in a local optimum without such a mechanism [90].

In practice, we can implement this mechanism at two different levels. First, the no-hope counters or sensors are placed in each particle to track the individual best so far (*pbest*). Second, we can set only one counter to monitor the entire swarm to track the status of the global best so far (*gbest*). Once a no-hope situation occurs, we restart the swarm and keep the best experience so far or refresh the best experience completely. We developed the *genericRestart* algorithm based on the continuous *genericPSO* algorithm. Similar to the no-hope & re-hope strategy in the BCSP model, *genericRestart* has a no-hope counter in each particle. Once a no-hope is detected, the algorithm refreshes the particles' individual best so far (*pbest*), but retains the global best so far (*gbest*) that continuously guides the swarm.

4.2.2.3 No-hope and Hop: genericHop, binaryHop, grayHop and bcspHop

Unless the **solution density** of a problem or the number of solutions in a unit area of the search space is relatively high, blindly and randomly restarting the swarm may not solve the problem effectively. Alternatively, we propose to incorporate a repair-based constraint solving method with the no-hope & re-hope mechanism, and have particles fix constraint violations locally to find a better assignment for the particle position. Specifically, once a no-hope situation has been detected, the particles *hop* in some dimensions where the variable assignments are in conflict the most, and randomly assign a domain value[16] to each of the

[15] A no-hope count is a predefined upper bound for the maximum number of iterations before a re-hope action may take place.

[16] We can also consider applying some heuristics to select values such as the min-conflict heuristic [64].

corresponding dimensions (or variables) to repair the feasibility of the potential solutions. We name these algorithms *genericHop*, *binaryHop*, *grayHop*, and *bcspHop*[17] in this research.

4.2.2.4 Piggy bank: genericMultigbest

In PSO, all potential solutions are evaluated by an objective function. According to the returned evaluation, PSO determines the quality of a potential solution. The original PSOs keep only one global best found so far $\langle gbest, xgbest \rangle$ and throw away all the other $\langle gbest', xgbest' \rangle$ with the same evaluation where $gbest' = gbest$ but $xgbest' \neq xgbest$. However, the global best in use $\langle gbest, xgbest \rangle$ may not be the best choice because different potential solutions with the same evaluation do not necessarily have the same probability of solving a problem. This is particularly true if the objective function is not effective enough to distinguish different potential solutions. At worst, some of these potential solutions may even lead to a dead end at a local optimum. Because of the *multimodal* nature of CSPs[18] and the quality of the chosen objective function, it can be difficult to determine which of the potential solutions with the same evaluation is really better. One remedy to this problem is to *bank* all these potential solutions with the same evaluation, and then use them one at a time when a no-hope situation is detected. Pragmatically, once a chosen *xgbest* cannot guide the swarm to get any improvement for a certain amount of time, the algorithm replaces it with another banked *xgbest'* and continues searching. Currently, the timing of replacing a banked global best is still at trial-and-error stage, so we only implement this particle swarm *genericMultigbest* in the Continuous model for the experiment.

4.2.2.5 Diversity control: genericAttractRepulse

Research points out that the original PSO may converge too fast to find an actual optimum for hard problems, particularly for multimodal problems [115] like CSPs. The no-hope and re-hope strategies discussed so far, are designed to help the swarm to escape from a local optimum. But, they only take place when a no-hope situation occurs. In that case, the swarm has probably already been trapped. Different from these no-hope and re-hope strategies, Vesterstrøm and Riget proposed a **diversity control** system to monitor the

[17] *genericHop* is based on the Continuous model, *binaryHop* and *grayHop* are on the Discrete model, and *bcspHop* is upon the BCSP model.

[18] CSPs typically have multiple solutions. For PSO, these multiple solutions mean multiple optima in the search space.

distance among the particles and actively prevent the swarm from converging too soon [115]. Their research has shown that the new system is capable of finding better optima than the traditional PSO in a continuous search space. However, with the time given for this research, we cannot find proper parameter settings for the *genericAttractRepulse* algorithm to work with integer domains; thus, we can only leave this system for future research.

4.2.2.6 Partner exchange: genericExchange

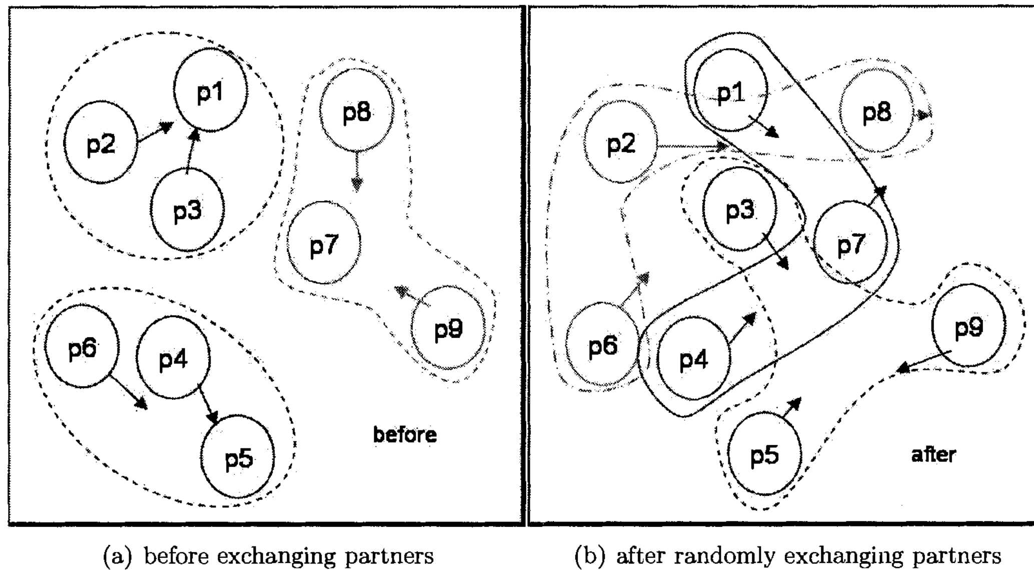
Fast convergence is a merit of PSOs in dealing with **unimodal problems** in which only one optimal solution exists, but fast convergence may cause the swarm to prematurely stop in solving multimodal problems such as CSPs [115]. As discussed in Section 3.3.1, the neighbourhood structures of the swarm affect the speed of convergence. So, we propose to arrange a particle's neighbours to maintain the diversity of the swarm and slow down the convergence. Specifically in the *genericExchange* algorithm, we have the particles exchange partners every so often across different neighbourhoods before the entire swarm converges. Suppose we have a swarm structured in a logical ring with $k = 2$ as shown in Figure 4.7, where each neighbourhood includes 3 particles. Before exchanging partners, we have particles (P_1, P_2, P_3), (P_4, P_5, P_6) and (P_7, P_8, P_9) as three groups in Figure 4.7(a). When some predefined criterion[19] arrives, the exchange partner mechanism takes place and we randomly swap the members of each neighbourhood as shown in Figure 4.7(b), for example. After the swap, particles update their velocities based on the new neighbourhood information.

4.2.2.7 Local depth-first search: genericDFS

If the swarm converges and the particles lack in power to fly around a local area,[20] there may be other solutions unvisited around the area. Such scenarios can be explored with a depth-first search (DFS) to systematically look for an optimum locally. However, relying on a complete search too much may not only defeat the purpose of using the swarm in the first place, but also prolong the search if the problem is too big. Therefore, we should keep the *genericDFS* algorithm under control while integrating a depth-first search. For example, we divide all n CSP variables into several groups to keep the local DFS manageable. Further

[19]For the experiment, we define a counter and have the swarm perform the strategy periodically when the counter increments at certain times.

[20]*gbest* and *pbest* cannot give a particle enough velocity to move around the local area thoroughly.



(a) before exchanging partners (b) after randomly exchanging partners

Figure 4.7: Exchanging partner is taking actions.

details on setting up the algorithm is available under Section A.3.1.

4.2.2.8 Hybrid PSO: genericHybrid

All the strategies discussed so far have different characteristics and contribute to the search differently. Enabling the application of different strategies at different situations, the hybrid PSO (*genericHybrid*) integrates the no-hope and hop, partner exchange and local depth-first search strategies described in the previous sections. In addition, our early experiments showed that the more particles the more effectively the swarm performs in general;[21] or at least, there may be a better chance to get good "guessers" who can produce good initial solutions. Therefore, we also enable the *genericHybrid* algorithm to **spawn** or generate more particles when there is no hope for improving the solutions. However, we do not rely on spawning more particles to work on its own because the performance of a generic swarm does not become better by simply increasing the number of particles for complex problems. Besides, the run time of each iteration grows as the number of particles increases. Therefore, we only allow the swarm to spawn more particles when a serious stagnation in the progress of

[21]This is only a general comment to some extent, not a conclusion. More experiments should be done.

the algorithm is detected. When the no-hope and re-hope strategies cannot help to improve the solutions and the swarm obviously needs more alternative solution such as the doubling of the no-hope count, the swam then spawns more particles to continue searching.

4.2.3 Neighbourhood structures

In this research, all the particle swarm algorithms are implemented with three different neighbourhood structures. Besides a star and a ring that are adopted from the traditional PSOs, we mixed the two to have a **star-ring** structure. In a star neighbourhood, a particle uses the global best information (*gbest* and *xgbest*) and its own experience (*pbest* and *xpbest*) in decision-making. In a ring, a particle updates its velocity and position according to its neighbours' (local) best information (*lbest* and *xlbest*) and its own experience (*pbest* and *xpbest*). In a star-ring structure, a particle works with two different neighbourhood structures. Not only does it globally retrieve information from all the particles in a star, but it also interacts with its local neighbourhoods[22] in rings. Computationally, *xgbest*, *xlbest* and *xpbest* all contribute to particles' velocity updates as follows:

$$\vec{v_i}(t) = \vec{v_i}(t-1) + r_1c_1(xpbest_i - x_i(t-1)) + r_2c_2(xgbest - x_i(t-1)) + r_3c_3(xlbest_k - x_i(t-1)) \tag{4.1}$$

where r_1, r_2 and r_3 are random numbers between 0 and 1, and c_1, c_2 and c_3 are the acceleration constants.[23]

4.2.4 Summary of particle swarm algorithms

Table 4.1 summarizes all the particle swarm algorithms that have been implemented as part of this research. Aside from the algorithms based on the BCSP model, the rest of them use the two objective functions discussed in Section 4.1.2 to guide the search. These two objective functions, the conflict count function and the distance estimation function, are used individually with each algorithm and not in conjunction.

[22] A particle may belong to multiple local neighbourhoods as described in Section 3.3.1.

[23] Acceleration constants are discussed in Section 3.3.2 for a star or a ring neighbourhood structure.

Table 4.1: This table summarizes the PSO algorithms developed in this research.

Algorithm	Model	Strategy	Strategy description	Section
genericPSO binaryPSO grayPSO bcspPSO	Continuous Discrete Discrete BCSP	generic	The original algorithm with minimal modifications to prevent the particles from falling out of CSP domains.	4.2.1.1 4.2.1.2 4.2.1.2 4.2.1.3
genericZigzag binaryZigzag grayZigzag bcspZigzag	Continuous Discrete Discrete BCSP	zigzag	Instead of traditionally moving in all n dimensions at once, these zigzag particles move one dimension at a time.	4.2.2.1 4.2.2.1 4.2.2.1 4.2.2.1
genericRestart	Continuous	no-hope& rehope	Once a no-hope is detected, the swarm restarts and brings more hopes to the system.	4.2.2.2
genericHop binaryHop grayHop bcspHop	Continuous Discrete Discrete BCSP	no-hope & hop	Integrating the no-hope& re-hope mechanism and a repair-based method, the particles try to fix constraint violations in the dimensions that have the most number of conflicts.	4.2.2.3 4.2.2.3 4.2.2.3 4.2.2.3
genericMultigbest	Continuous	piggy bank	The swarm keeps multiple global best found so far of the same evaluation and replaces the current best with a banked one when a no-hope situation occurs.	4.2.2.4
genericAttract-Repulse	Continuous	diversity control	The swarm monitors the distance among particles and actively prevents the swarm from prematurely converging.	4.2.2.5
genericDFS zigzagDFS	Continuous Continuous	DFS zigzag + DFS	When some predefined criterion arrives, each particle performs a complete depth-first search over their own responsible variables on the best solution found so far.	4.2.2.7 4.2.2.1+ 4.2.2.7
genericExchange zigzagExchange	Continuous Continuous	partner exchange partner exchange + zigzag	The swarm exchanges partners among different neighbourhoods periodically; after each swap, the particles compute their velocities based on the newly formed neighbourhood experience.	4.2.2.6 4.2.2.1+ 4.2.2.6
genericHybrid	Continuous	no-hope& hop+ DFS + spawned particles+ partner exchange	In addition to the features in genericHop, genericDFS and genericExchange, it includes spawning particles when the swarm is seriously stagnant (e.g. twice of the no-hope count)	4.2.2.3+ 4.2.2.6+ 4.2.2.7

4.3 Application Problem—PC Configuration

Random constraint satisfaction problems are often used for benchmarking algorithms, but they may not be practical [32]. Thus, rather than experimenting with many random constraint satisfaction problems, we will evaluate the performance of the algorithms on more realistic problems.

4.3.1 Introduction

Configuration problems are a class of problems related to considering how to assemble a product from a set of components under a set of limitations [103]. It includes a wide range of complex real-life problems such as equipment configuration, product configuration, network configuration, software configuration and service configuration. Because of the nature of the constraints of the problems, much problem solving research is focused on reasoning approaches [84] or constraint-based methods [49].

A PC configuration problem is a practical configuration problem, which configures computer parts to build a functional personal computer. This problem can be very complex and involves many kinds of hardware specifications. For example, certain CPUs can only fit into certain motherboards, for which the sockets must be the same, the frequency must be compatible, and so on. A basic barebone PC may easily involve twelve essential parts and a number of limitations to put them together. Besides, we also need to consider consumers' budget and preference. Solving such a problem can not only help computer stores to assemble computers based on the hardware specifications efficiently, but also assist both power users and naive users to purchase computer systems effectively with their preference taken into account. The PC configuration CSP [104] lends itself as a good candidate for our experimentation. As the purpose of this research focuses on solving CSPs as opposed to optimizing them, all constraints will be treated as *hard constraints* that must be satisfied concurrently. The test problems are modelled in the CSP Python framework [20] and used to evaluate the particle swarm algorithms described earlier.

4.3.2 Modelling a PC configuration problem in Python CSP Framework

To model the PC configuration problem, a set of variables, their domain values, and a set of constraints need to be defined. A PC configuration problem deals with a set of hardware components and each component has a set of specifications. Hardware components include

CPUs, memories, motherboards, interface cards and various peripheral items. Hardware compatibility and user preference are the two major types of constraints. For instance, CPU socket must fit into a motherboard, memory pins have to match, and the interface cards should be supported. Manufacturers, colour and monitor size are examples of common user requirements.

By analyzing the problem with the sample data taken from the Internet,[24] we consider two formulations below. Each of them models the problem to different levels of depth.

4.3.2.1 Simple formulation – Formulation I

Our first formulation is similar to Tam and Ma's model for building a web-based configuration application [104, 105]. The categories of the hardware components are identified as the problem variables. Component compatibility and user requirements are preprocessed and turned into good list constraints.

The benefit of this formulation is that the search space is much smaller and the problem is much simpler than the second formulation introduced in Section 4.3.2.2. Much work has already been done during the preprocessing phase. For example, a number of constraints between CPU and motherboard such as (CPU socket vs. motherboard socket) and (CPU speed vs. motherboard frequency) can be reduced to one good list constraint. On the other hand, it is not as user friendly in terms of formulating a CSP. It requires constraint preprocessing among different components and does not fully take advantage of CSP representation. Moreover, it has to preprocess the constraints once again if more components or constraints are added later on, and it is less obvious how to represent user requirements.

4.3.2.1.1 CSP variables and the domains. In this formulation, we define only a set of **component variables** to represent the categories of the computer components such as var_{cpu}, var_{ram} and var_{mb} for CPU, RAM and motherboards. Figure 4.8 illustrates the variables of a sample problem.

To represent the sample components as a set of integers for each component variable, we enumerate the components of the variable. For example, 10 CPU components are listed in Table 4.2. These CPU components are numbered from 0 to 9 and the domain D_{cpu} is set to $\{0, 1, 2, \ldots, 9\}$. Likewise, the domains of var_{ram} and var_{mb} are set to $\{0, 1, 2, \ldots, 9\}$

[24]The components are partially chosen from the hot sale list of http://www.ncix.com on October 2004 and the specifications are gathered from various manufacturers' online information.

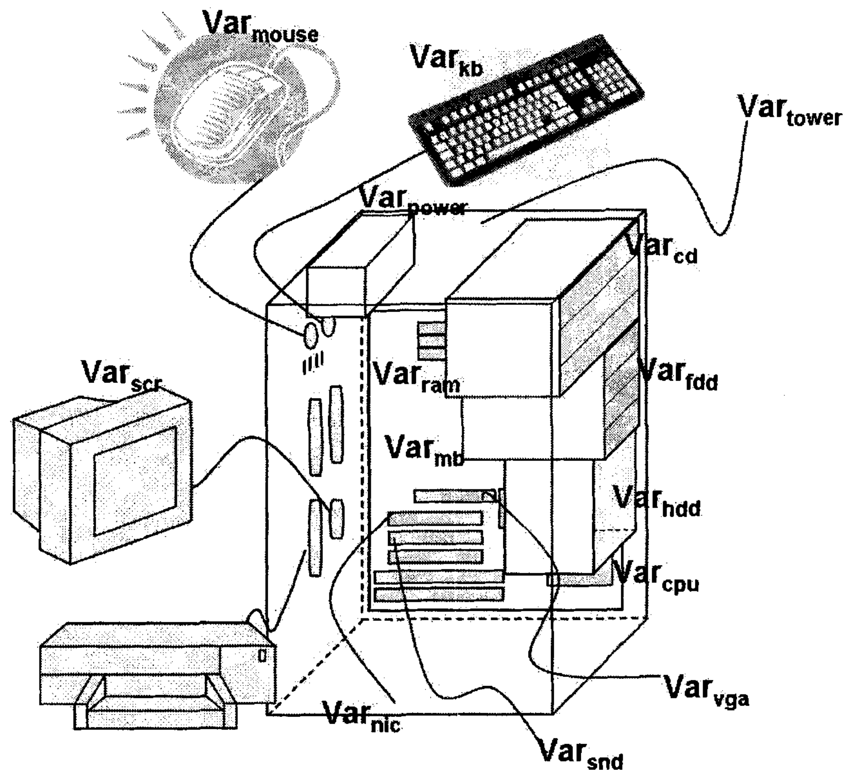


Figure 4.8: The CSP variables of a PC configuration problem under Formulation I.

representing the RAM and motherboards listed in Table B.2 and Table B.3 respectively. For the rest of the sample components and their domain assignments, see Section B.1.1.

4.3.2.1.2 CSP Constraints. Tam and Ma assume the availability of a preprocessed, centralized database is available and the data is consistent [104]. The only constraints explicitly considered are on user budget, CPU type and memory bus speed; all other components are not constrained. When computers are configured however, hardware compatibility alone is in fact much more complicated than only CPU and memory constraints. Although this research is not about how to represent a configuration problem, we want to make the test problem reasonably realistic; thus, we add additional realistic constraints to model the problem. For example, several constraints among CPU, RAM and motherboard are listed in Table 4.3. For the rest of the constraints in Formulation I, see Table B.16 in Section B.1.2. In

Table 4.2: Sample CPUs for var_{cpu} and their integer representation.

Component specification	Enumeration
AMD ATHLON 64 3000+ 2.0GHz S754 800fsb	0
AMD Mobile ATHLON XP-M 2500+ 1.86GHz SOCKETA 266fsb	1
INTEL PENTIUM 4 3.0GHz S478 800fsb	2
AMD ATHLON 64 3200+ 2.2GHz S754 800fsb	3
AMD ATHLON 64 3500+ 2.2GHz S939 2000fsb	4
AMD SEMPRON 2500+ 1.75GHz SOCKETA 333fsb	5
INTEL PENTIUM 4 2.8GHz S478 800fsb	6
INTEL PENTIUM 4 3.0GHz S478 800fsb	7
AMD ATHLON 64 2800+ 1.8GHz S754 1600fsb	8
INTEL PENTIUM 4 3.2GHz S478 800fsb	9

The domain of var_{cpu} is the enumeration of the sample components in $\{1, 2, \ldots, 9\}$.

this formulation, we preprocess the data into good list constraints on a number of variables to enforce corresponding constraints.

Table 4.3: Sample constraints on var_{cpu}, var_{ram} and var_{mb} under Formulation I.

Constraint	Description
GOOD(var_{cpu}, var_{mb})	CPU socket must fit on a motherboard; fsb (front side bus) should be compatible.
GOOD(var_{mb}, var_{ram})	memory pins and the slots on a MB have to match; if RAM is a dual RAM, motherboard must support it
GOOD(var_{cpu}, var_{ram}, var_{mb})	total price $\leq$ budget

4.3.2.1.3 Formulation I in the Python CSP Framework. To implement this formulation in the Python CSP Framework [20], we extend the Framework and define one constraint for each compatibility limitation. Without any **specification variables**[25] however, there is not a clear way to represent these constraints descriptively. Besides, it is fairly inefficient to reprocess the component specification strings such as "CPU INTEL Pentium 4 2.8GHz SocketS478 800fsb" and "MB ASUS SocketS478 dualRAM 184pin (800,533,400)fsb"

[25] A set of variables represents the specifications of the hardware components. Refer to Section 4.3.2.2 for detail.

Table 4.4: Four n-ary constraints are added to the Python CSP Framework for Formulation I.

Constraint	arity	Condition
`GOODlist`	n	if the assignment is on the list, returns true
`BADlist`	n	if the assignment is not on the list, returns true
`UPPERprice`	n	if "total price $\leq$ user budget", returns true
`LOWERprice`	n	if "total price $\geq$ user budget", returns true

"Arity" indicates the number of arguments of a constraint. If the condition is true, the constraint will return TRUE.

in every constraint check. Thus, most of the component constraints shown in Section B.1.2 are preprocessed into corresponding good lists or bad lists, except for the user budget. Four n-ary constraints: `GOODlist`, `BADlist`, `UPPERprice` and `LOWERprice` listed in Table 4.4, are defined in the Framework.

4.3.2.2 Detailed formulation – Formulation II

Besides the component variables in the Formulation I, we can model the problem more descriptively with a set of **specification variables**. These specification variables represent the specifications of hardware components such as brand name, model, capacity, and so on. Together with the Python CSP Framework, this formulation interestingly describes **connection constraints** and **user constraints** for component compatibility and user requirements.

4.3.2.2.1 CPS variables and the domains. In addition to the component variables that represent the categories of the computer components: *cpu*, *ram*, *mb*, etc., we further define a set of specification variables to describe the specifications of the components such as cpu_{brand}, cpu_{model}, cpu_{socket} for a CPU. Figure 4.9 shows the variables of a sample problem in Formulation II.

Similar to the domains in Formulation I, the components of each category are enumerated as the integer domain values for the corresponding component variable. The values of each component specification are also enumerated as the domain values for the corresponding specification variable. Taking the sample CPUs in Table 4.2, we have specification variables cpu_{brand}, cpu_{model}, cpu_{clock}, cpu_{socket} and cpu_{fsb} for the the brand name, model,

CPU clock, socket and fsb.[26] Table 4.5 illustrates how to enumerate the values of individual specifications. For the rest of the sample specifications, see Section B.2.1.

Table 4.5: Sample values of CPU specifications and the enumerated domain.

enum	cpu_{brand}	enum	cpu_{model}	enum	cpu_{clock}	enum	cpu_{socket}	enum	cpu_{fsb}
0		0	ATHLON	0	1.75	0	S478	0	266
1	AMD	1	M ATHLON[27]	1	1.8	1	S754	1	333
2	INTEL	2	PENTIUM	2	1.86	2	S939	2	800
3		3	SEMPRON	3	2.0	3	SOCKETA	3	1600
4		4		4	2.2	4		4	2000
5		5		5	2.8	5		5	
6		6		6	3.0	6		6	
7		7		7	3.2	7		7	

The rows in the tables do not represent a product, but the enumerated values. For example, AMD of cpu_{brand} is 1, ATHLON of cpu_{model} is 0, PENTIUM of cpu_{model} is 2, SOCKETA of cpu_{socket} is 3, and so on.

4.3.2.2.2 CSP Constraints. We define three types of constraints in this formulation: component constraints, connection constraints and user constraints. **Component constraints** relate specification variables to their corresponding component variables in forms of good lists to describe individual hardware components. For instance, to enforce "CPU: AMD ATHLON 64 3000+ 2.0GHz S754 800fsb", we lookup the CPU from Table 4.2 and its specifications from Table 4.5. Consequently, the CPU has a value $cpu = 0$ in Table 4.2. By parsing the specifications and looking up the corresponding values in Table 4.5, we get $cpu_{brand} = 1$, $cpu_{model} = 0$, $cpu_{clock} = 3$, $cpu_{socket} = 1$ and $cpu_{fsb} = 2$, respectively. We can then put these values in a tuple $(cpu, cpu_{brand}, cpu_{model}, cpu_{clock}, cpu_{socket}, cpu_{fsb}) = (0, 1, 0, 3, 1, 2)$. After processing all the sample CPUs collectively, we have a 6-ary component constraint GOODcpu(cpu, cpu_{brand}, cpu_{model}, cpu_{clock}, cpu_{socket}, cpu_{fsb}) and a good list of CPUs as in Table 4.6. For the rest of component good lists under Formulation II, refer to Section B.2.2.1.

Connection constraints enforce component compatibility. Instead of using good lists as those in Formulation I, these constraints can be represented in arithmetic relations to work within the Python CSP Framework. Sample connection constraints are listed in Table 4.7. For a complete list of connection constraints, see Table B.39 in Section B.2.2.2.

[26] A specification of CPU, stands for front side bus and represents the speed.

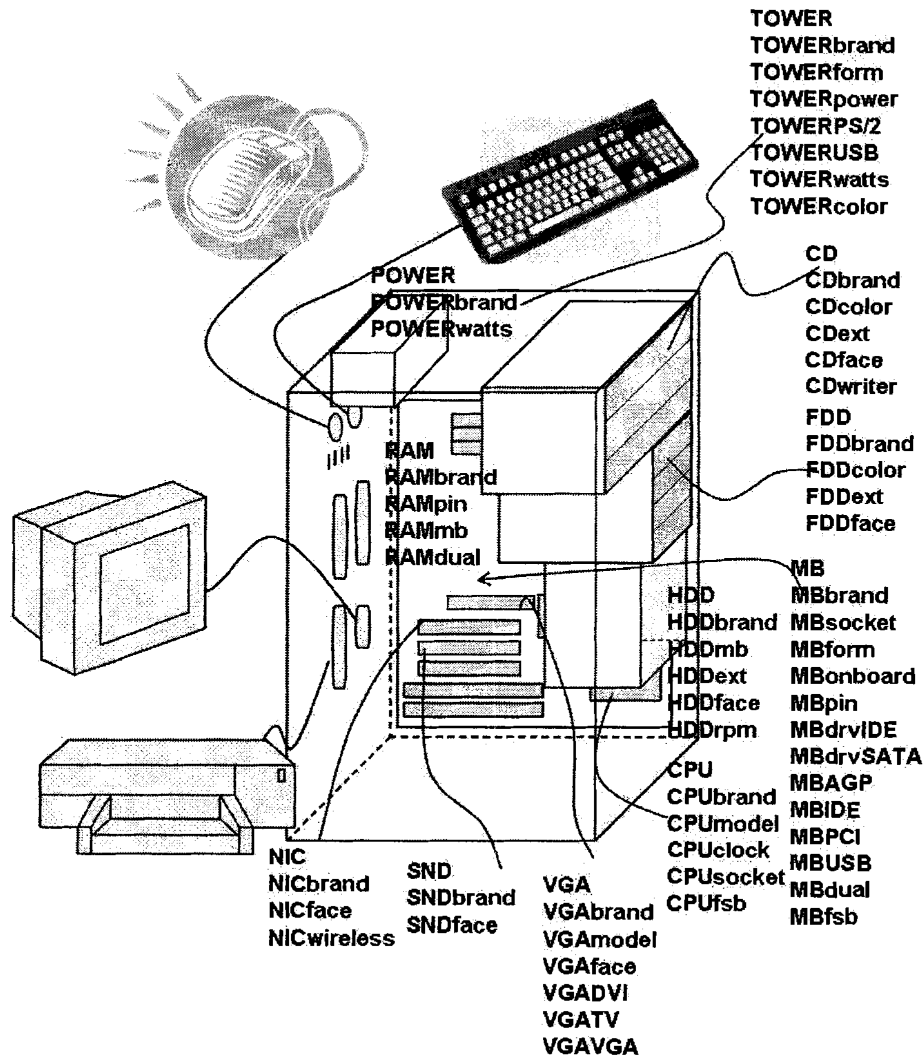


Figure 4.9: The CSP variables of a PC configuration problem under Formulation II.

Table 4.6: Sample CPUs in good tuples, and the entire list represents "GOODcpu" constraint.

$(cpu$	cpu_{brand}	cpu_{model}	cpu_{clock}	cpu_{socket}	$cpu_{fsb})$
(0,	1,	0,	3,	1,	2)
(1,	1,	1,	2,	3,	0)
(2,	2,	2,	6,	0,	2)
(3,	1,	0,	4,	1,	2)
(4,	1,	0,	4,	2,	4)
(5,	1,	3,	0,	3,	1)
(6,	2,	2,	5,	0,	2)
(7,	2,	2,	6,	0,	2)
(8,	1,	0,	1,	1,	2)
(9,	2,	2,	7,	0,	2)

Table 4.7: Sample PC connection constraints in Formulation II.

<table>
<tr><th>Constraint expression</th><th>arity</th><th>Description</th></tr>
<tr><td><code>CPU_socket == MB_socket</code></td><td>2</td><td>CPU socket must fit on a motherboard</td></tr>
<tr><td><code>RAM_pin == MB_pin</code></td><td>2</td><td>memory pins and the slots on a MB have to match</td></tr>
<tr><td><code>(SND_brand != 0) | (MB_snd != 0)</code></td><td>2</td><td>if a MB has a sound chip, a sound card is optional</td></tr>
<tr><td><code>SND_face < 4</code> and
<code>((((SND_face+1) * (MB_PCI*100))</code>
<code>/ 10**SND_face) % 10) > 0</code></td><td>1
3</td><td>if a sound card is to use, the interface must be supported</td></tr>
</table>

User constraints are defined for user requirements or preference. The requirements can be varied and sample user constraints are listed in Table 4.8.

4.3.2.2.3 Formulation II in the Python CSP Framework. In addition to the `GOODLIST`, `BADLIST`, `UPPERprice` and `LOWERprice` constraints (shown in Table 4.4), a special arithmetic expression '`|`' for '`OR`' have been implemented. Several connection constraints have been described using such an expression such as `(SND_brand != 0) | (MB_snd != 0)` (from Table 4.7). This expression is interesting because traditionally all CSP constraints are logically handled with `AND` relations and all of them must be satisfied at the same time.

Table 4.8: Sample PC user constraints in Formulation II.

Constraint expression	**arity**	**Description**
`UPPERprice(items, 1800)`	n	budget upper bound $1800
`LOWERprice(items, 1500)`	n	price lower bound $1500
`FDD_external == 1`	1	want to have an external floppy drive
`CD_writer != 1`	1	do not want a CD writer

However, in reality some constraints may be satisfied in one way or another. The `OR` expression enhances the flexibility of the Framework in expressing constraints and the user can have an option to specify more varieties of constraints in the Framework.

4.3.3 PC configuration test problems

In order to test the particle swarm algorithms on both Formulation I and II, different sets of test problems have been defined.

4.3.3.1 Problems for Formulation I

A total of 6 test problems have been defined under Formulation I. These problems are listed in Table 4.9. Among these problems, problem 10.3 and problem 10.41 are for overall experiments and the others focus on dealing with n-ary price constraints. See Section B.1.3 for problem description of each problem.

4.3.3.2 Problems for Formulation II

A total of 14 test problems have been defined under Formulation II. These problems are divided into seven problem sets 20, 21, ... 26 listed in Table 4.10. The complexity of these problem sets progressively increases. Each problem set contains a base problem and each problem of a problem set is developed by incrementally adding a number of constraints. See Section B.2.3 for problem description of each problem.

Table 4.9: PC configuration problems for Formulation I.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	approx. density
10.3	14	10 – 40	1.01×10^{15}	14	2	14	1.1×10^{-1}
10.41	14	10 – 40	1.01×10^{15}	17	2	14	7.4×10^{-5}
10.53	14	10 – 40	1.01×10^{15}	16	2	14	(note: 1)
10.55	14	10 – 40	1.01×10^{15}	16	2	14	-
10.62	14	10 – 40	1.01×10^{15}	16	2	14	no solution
10.64	14	10 – 40	1.01×10^{15}	16	2	14	no solution

1. It takes too much time to estimate. After running the estimation program for days, we still cannot retrieve the result. Based on the number of nodes in the search space examined, we know the solution density of problem 10.53 is smaller than 2.0×10^{-7}. The solution density of problem 10.55 is not available either.
2. The 14 constraints in problem 10.3 are all connection constraints for compatibility. The additional constraints added to problem 10.41 relate to user requirements: one for external colour and the other two for the budget upper bound and the lower bound.

Table 4.10: PC configuration problems for Formulation II.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	approx. density
20.3	32	2 – 40	9.81×10^{16}	20	1	12	(note: 1)
21.3	36	2 – 40	5.89×10^{18}	20	1	12	-
22.3	46	2 – 40	3.62×10^{23}	23	1	12	-
23.3	51	2 – 40	1.16×10^{25}	23	1	14	-
24.3	58	2 – 40	3.36×10^{29}	31	1	14	-
25.3	59	2 – 40	6.72×10^{29}	31	1	14	-
25.43	59	2 – 40	6.72×10^{29}	35	1	14	-
25.47	59	2 – 40	6.72×10^{29}	39	1	14	-
25.50	59	2 – 40	6.72×10^{29}	45	1	14	-
25.53	59	2 – 40	6.72×10^{29}	33	1	14	-
25.55	59	2 – 40	6.72×10^{29}	33	1	14	-
25.62	59	2 – 40	6.72×10^{29}	33	1	14	no solution
25.64	59	2 – 40	6.72×10^{29}	33	1	14	no solution
26.3	70	2 – 40	1.14×10^{37}	31	1	15	-

1. It takes too much time to estimate. After running the estimation program for days, we still cannot retrieve the result. Based on the number of nodes in the search space examined, we know the solution density of problem 20.3 is smaller than 2.5×10^{-7}. None of the others is available either.

Chapter 5

Experiment and Evaluation

5.1 Introduction

In Section 3.6, we stated our research questions. In Chapter 4, we described a number of new particle swarm algorithms developed for this research. In this chapter, we focus on the answers. In support of our answers, we empirically tested the new algorithms in three phases, the Exploration, Comparison and `all_diff` phases. In the Exploration phase, we explored the algorithms and their parameter settings, and determined which algorithms and what parameter settings for the Comparison phase. In the Comparison phase, we tested the algorithms on PC configuration test problems and collected the data for comparing the effectiveness and efficiency of the algorithms. From the Exploration and Comparison phases, we realized that the particle swarm algorithms have difficulty in solving n-ary constraints for large n, and so we set up the `all_diff` experiment to examine the observation. Based on the results, we evaluated the algorithms on a set of measures.

This chapter begins with the presentation of the test problems and the measures. In Section 5.3, the use of facilities and the implementation issues of the algorithms are explained. In Section 5.4, the experimental results are reported and analyzed. Finally in Section 5.5, based on the experimental results, the research questions stated in Section 3.6 are addressed.

5.2 Experiment Setup

5.2.1 Test algorithms

In order to solve CSPs, we developed new particle swarm algorithms based on three original PSOs: the continuous PSO [53], binary discrete PSO [54] and Schoofs and Naudts' PSO for solving binary CSPs (BCSP) [90]. We refer to these particle swarm algorithms as the Continuous model, Discrete model and BCSP model with respect to their origins. Table 4.1 provides a list of algorithms and their description. The development of our algorithms has been explained in Section 4.2, and each of these algorithms has been designed to work with different neighbourhood structures to communicate globally, locally or both.[1] Our research focuses on constraint satisfaction problems rather than on constraint optimization problems; therefore, all constraints must be satisfied simultaneously.

Except for the BCSP-based algorithms that are already designed to work with the conflict count objective function, the rest of them use the two objective functions discussed in Section 4.1.2 to guide the search. These two objective functions, the conflict count function and the distance estimation function, are used individually with each algorithm and not in conjunction. We divide the algorithms into five classes according to their original models and the objective functions, and list them in Table 5.1.

Table 5.1: The five classes of particle swarm algorithms in this research.

Model	Objective function	Referred name
Continuous	conflict counts	Continuous-Conflict
Discrete	conflict counts	Discrete-Conflict
BCSP	conflict counts	BCSP
Continuous	distance estimation	Continuous-Distance
Discrete	distance estimation	Discrete-Distance

The *Continuous-Conflict* algorithms are derived from the Continuous model with conflict counts objective function.

In this research, we seek to propose several particle swarm algorithms which are good for solving general CSPs, so we do not emphasize on tuning parameter settings to improve the

[1]We denote each of these as "pg", "pl" and "plg"; with these structures, particles take information from the global best *gbest* and the individual best *pbest*, or from the local best *lbest* and the individual best *pbest*, or from the global best *gbest*, the local best *lbest* and the individual best *pbest* respectively.

performance of the algorithms. Thus, most of the parameter settings for the experiments are obtained from previous particle swarm research. Anyone interested in the effects of the parameters may further tweak the algorithms as needed for their particular problem.

5.2.1.1 Algorithms for the Exploration phase

In the Exploration phase, we examined all the algorithms shown in Table 5.2.[2] For each algorithm, we set the swarm size to 20, 50 and 100 with an iteration limit of 10000. To experiment with neighbourhood structures, we have the swarm set up to work with a global neighbourhood structure, a local neighbourhood structure of size 7 (i.e. $k = 6$), and a mix of both structures. The detailed settings of the algorithms are in Table C.1 in Section C.1 and the parameters used to control the behaviour of the swarm are explained in Table 5.3.

Table 5.2: The PSO algorithms used in the Exploration phase.

model-objective	algorithm	sect.	parameters in use
Continuous-conflict	genericPSO	4.2.1.1	ω^1, c_1 and $c_2{}^2$
	genericZigzag	4.2.2.1	ω, c_1 and c_2
	genericHop	4.2.2.3	ω, c_1 and c_2, *nohope*[3], *pop_rate*[4]
	genericRestart	4.2.2.2	ω, c_1 and c_2, *nohope*, *pop_prate*
	genericMultigbest	4.2.2.4	ω, c_1 and c_2, *nohope*
	genericDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*[5]
	zigzagDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*
	genericExchange	4.2.2.6	ω, c_1 and c_2, *regroup*[6], *stop_group*[7]
	zigzagExchange	4.2.2.6	ω, c_1 and c_2, *regroup*, *stop_group*
Discrete-conflict	binaryDiscrete	4.2.1.2	ϕ_1 and $\phi_2{}^9$, v_{max} and $v_{min}{}^{10}$
	grayDiscrete	4.2.1.2	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	grayZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*[11], *pop_rate*[12]
	grayHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
BCSP-conflict	bcspPSO	4.2.1.3	φ_1 and $\varphi_2{}^{13}$, *deflection*[14], *nohope*[15]
	bcspZigzag	4.2.2.1	φ_1, φ_2, *deflection*, *nohope*

[2]In Table 5.2, we also provide information about the section number and the related parameter settings for reference.

model-objective	algorithm	sect.	parameters in use
	bcspHop	4.2.2.3	φ_1, φ_2, *deflection*, *nohope*, *pop_rate*[16]
Continuous-distance	genericPSO	4.2.1.1	ω[1], c_1 and c_2[2]
	genericZigzag	4.2.2.1	ω, c_1 and c_2
	genericHop	4.2.2.3	ω, c_1 and c_2, *nohope*[3], *pop_rate*[4]
	genericRestart	4.2.2.2	ω, c_1 and c_2, *nohope*, *pop_prate*
	genericMultigbest	4.2.2.4	ω, c_1 and c_2, *nohope*
	genericDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*[5]
	zigzagDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*
	genericExchange	4.2.2.6	ω, c_1 and c_2, *regroup*[6], *stop_group*[7]
	zigzagExchange	4.2.2.6	ω, c_1 and c_2, *regroup*, *stop_group*
Discrete-distance	binaryDiscrete	4.2.1.2	ϕ_1 and ϕ_2[9], v_{max} and v_{min}[10]
	grayDiscrete	4.2.1.2	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	grayZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*[11], *pop_rate*[12]
	grayHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*

Table 5.3: Parameters used in the Exploration phase

model-objective	parameter	description
(across algorithms)	*pop*	the population of the swarm
	k	the size of neighbourhood
	$ITER$	the maximum of iterations
Continuous-conflict	ω	an inertia weight in computing particle's velocity, determines the effect of the previous velocity at time $t-1$
	(c_1,c_2)	the acceleration constants in computing particle's velocity, determines the influence of the global (or local) best information and the individual best information
	nohope	an iteration count, defines when the swarm has no more improvement for so long, the swarm performs a certain strategy to break the situation

model-objective	parameter	description
	pop_rate	the percentage of particles to perform the given strategy after the *nohope* count kicks in; for instance, *pop_rate* = 0.5 means half of the population should perform the specific strategy
	dfs_size	defines the number of variables in each depth-first search group assigned to a particle; see Section 4.2.2.8
	regroup	an iteration count, defines when the swarm should perform such a strategy; only used in algorithms involving "exchange partner" strategy
	stop_group	an iteration count, defines when to stop regrouping particles and return to normal
	spawn	an iteration count, defines when to spawn more particles; used in genericHybrid algorithm
Discrete-conflict	ϕ_1 and ϕ_2	the acceleration constants in computing particle's velocity
	(v_{max},v_{min})	the velocity upper bound and lower bound in determining particle's velocity
	nohope	same as in the Continuous
	pop_rate	same as in the Continuous
BCSP-conflict	φ_1, φ_2	the coefficients are used in computing particle's velocity
	deflection	serves as a switch to refine particle's moving direction, i.e. whether a particle should flip the direction or not
	nohope	exists in the original BCSP model for individual particles to determine when it has done no improvement and should restart; we also use this parameter globally to the swarm similar to the *nohope* in the Continuous model
	pop_rate	same as in the Continuous
Continuous-distance		same as Continuous/conflict
Discrete-distance		same as Discrete/conflict

5.2.1.2 Algorithms for the Comparison phase

Based on the results from the Exploration phase, several algorithms were eliminated for two reasons. One, we could not find proper parameter settings for those algorithms within the

time constraint. Also, in order to include additional algorithms (described below) that we would like to examine within the limited time for this research we removed several algorithms from the experiment.

Additional algorithms are developed and included by integrating the strategies we have examined. "zigzagHop" type algorithms including "zigzag" style movement (as described in Section 4.2.2.1) and a repair-based "no-hope and hop" strategy (as discussed in Section 4.2.2.3) are added. Each of these zigzagHop algorithms will be referred to as *genericZigzagHop* of the Continuous PSO, *binaryZigzagHop* and *grayZigzagHop* of the Discrete PSO with different encodings, and *bcspZigzagHop* of the BCSP PSO. Another algorithm added is *genericHybrid*, which integrates the "no-hope and hop", "partner exchange", and "local depth-first search" strategies as discussed in Section 4.2.2.8. Table 5.4 lists the algorithms used in this phase.

As for the parameter settings, we only used a subset of those from the Exploration phase because it is outside the scope of this research to exhaustively run all the settings on the selected algorithms. Some parameter settings; however, have been adjusted. First, we find that swarm size 20, 50 and 100 are more than sufficient for the purposes of the experiment. More particles imply more processing time for each iteration. Some experiments showed that we can differentiate the algorithms in the experiment with fewer particles, so we reduced the swarm size to 3, 5 and 10.

Second, with the time we save by reducing the particles, we increased the iteration limit to 20000. In addition, the results from the Exploration phase indicated that zigzag type algorithms complete an iteration quickly because they only process one dimension each time. Given approximately the same amount of time, we can assign a larger iteration limit to these algorithms. Therefore, we allow zigzag and zigzagHop type algorithms to run for a maximum of 50000 iterations.

Table 5.4: PSO algorithms used in the Comparison phase.

model-objective	algorithm	sect.	parameters in use
Continuous-conflict	genericPSO	4.2.1.1	ω, c_1 and c_2
	genericZigzag	4.2.2.1	ω, c_1 and c_2
	genericHop	4.2.2.3	ω, c_1 and c_2, *nohope*, *pop_rate*
	genericZigzagHop[1]		ω, c_1 and c_2, *nohope*, *pop_rate*
	zigzagDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*
	zigzagExchange	4.2.2.6	ω, c_1 and c_2, *regroup*, *stop_group*
	genericHybrid	4.2.2.8	ω, c_1, c_2, *nohope*, *pop_rate*, *dfs_size*, *regroup*, *stop_group*
Discrete-conflict	binaryDiscrete	4.2.1.2	ϕ_1 and ϕ_2, v_{max} and v_{min}
	grayDiscrete	4.2.1.2	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	grayZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	grayHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	binaryZigzagHop[2]		ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	grayZigzagHop[3]		ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
BCSP-conflict	bcspPSO	4.2.1.3	φ_1 and φ_2, *deflection*, *nohope*
	bcspZigzag	4.2.2.1	φ_1, φ_2, *deflection*, *nohope*
	bcspHop	4.2.2.3	φ_1, φ_2, *deflection*, *nohope*, *pop_rate*
	bcspZigzagHop[4]		φ_1, φ_2, *deflection*, *nohope*, *pop_rate*
Continuous-distance	genericPSO	4.2.1.1	ω, c_1 and c_2
	genericZigzag	4.2.2.1	ω, c_1 and c_2
	genericHop	4.2.2.3	ω, c_1 and c_2, *nohope*, *pop_rate*
	genericZigzagHop[1]		ω, c_1 and c_2, *nohope*, *pop_rate*
	zigzagDFS	4.2.2.7	ω, c_1 and c_2, *nohope*, *dfs_size*
	zigzagExchange	4.2.2.6	ω, c_1 and c_2, *regroup*, *stop_group*
	genericHybrid	4.2.2.8	ω, c_1, c_2, *nohope*, *pop_rate*, *dfs_size*, *regroup*, *stop_group*
Discrete-distance	binaryDiscrete	4.2.1.2	ϕ_1 and ϕ_2, v_{max} and v_{min}
	grayDiscrete	4.2.1.2	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	grayZigzag	4.2.2.1	ϕ_1, ϕ_2, v_{max}, v_{min}
	binaryHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	grayHop	4.2.2.3	ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	binaryZigzagHop[2]		ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*
	grayZigzagHop[3]		ϕ_1, ϕ_2, v_{max}, v_{min}, *nohope*, *pop_rate*

1. *genericZigzagHop* algorithm combines the strategies used in *genericZigzag* and *genericHop* algorithms.
2. *binaryZigzagHop* algorithm combines the strategies used in *binaryZigzag* and *binaryHop.*
3. *grayZigzagHop* algorithm combines the strategies used in *grayZigzag* and *grayHop.*
4. *bcspZigzagHop* algorithm combines the strategies used in *bcspZigzag* and *bcspHop.*

Third, we also changed the neighbourhood structure settings for the experiment. Among the three neighbourhood structures, the local neighbourhood structure (pl) performs better than the other two in the Exploration phase. Much research in the field also use similar structures only. Taking the time limitation for this research into account, we choose the local neighbourhood structure for the Comparison phase experiment. Owing to the change of the swarm size, we modify the size of a neighbourhood to 3 (i.e. $k = 2$) correspondingly. See Table C.2 for the detailed algorithm settings.

5.2.1.3 Algorithms for the all_diff phase

The Exploration and Comparison phases of the experiment indicated our particle swarms have difficulty dealing with n-ary constraints for large n, such as the price constraints in the PC configuration problem and the `all_diff` constraints in an n-queens problem. These constraints become more difficult to the swarm as n grows. One obvious reason is that the particles lose their ability to distinguish the **culprit variables** whose assignments are the actual cause of the constraint violations. To observe the ability of the algorithms to handle n-ary constraints, we tested the same algorithms used in the Comparison phase with the same parameter settings on a set of n-queens problems for the `all_diff` phase. With the n-ary `all_diff` constraint in n-queens problems, we can systematically increment the size n and observe the results of the experiment. See Table 5.4 and Table C.2 for a list of algorithms and the parameter settings respectively.

5.2.2 Test problems

5.2.2.1 Test problems for the Exploration phase

In the Exploration phase, we tested the particle swarm algorithms on a number of basic PC configuration problems[3] and n-queens problems to explore the algorithms and their parameter settings, and so we could determine the algorithms and the parameter settings for the Comparison phase. The problem specifications are listed in Table 5.5 and Table 5.6. Refer to Section B.1.3 and Section B.2.3 for detailed problem descriptions.

[3] These include both Formulation I and Formulation II. Formulation I has been discussed in Section 4.3.2.1 and Formulation II can be found in Section 4.3.2.2

Table 5.5: PC configuration problems in the Exploration phase.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	approx. density
10.3	14	10 – 40	1.01×10^{15}	14	2	14	1.1×10^{-1}
10.40	14	10 – 40	1.01×10^{15}	15	2	14	3.6×10^{-4}
20.3	32	2 – 40	9.81×10^{16}	20	1	12	-
21.3	36	2 – 40	5.89×10^{18}	20	1	12	-
22.3	46	2 – 40	3.62×10^{23}	23	1	12	-
23.3	51	2 – 40	1.16×10^{25}	23	1	14	-
24.3	58	2 – 40	3.36×10^{29}	31	1	14	-
25.3	59	2 – 40	6.72×10^{29}	31	1	14	-
26.3	70	2 – 40	1.14×10^{37}	31	1	15	-

Problem 10.3 and 10.40 are under Formulation I, and the rest are under Formulation II.

Table 5.6: n-queens problems in the Exploration phase.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	solution #	solution density
4-queens	4	4	256	13	2	4	2	7.81×10^{-3}
5-queens	5	5	3125	21	2	5	10	3.20×10^{-3}
6-queens	6	6	46656	31	2	6	4	8.57×10^{-5}
7-queens	7	7	823543	43	2	7	40	4.86×10^{-5}
8-queens	8	8	16777216	57	2	8	92	5.48×10^{-6}

5.2.2.2 Test problems for the Comparison phase

To evaluate the algorithms on a variety of problems, we create more test problems by adding more constraints to each problem set,[4] in addition to the basic test problems in the Exploration phase. The problem specifications are summarized in Table 5.7. Refer to Section B.1.3 and Section B.2.3 for detailed problem descriptions.

The special test problems, numbered 53, 55, 62, and 64[5] that contain more difficult price constraints, require additional explanation. For these problems, the distance objective function has been modified to evaluate the price constraint and return a value between 1 and 0. If the return value is greater than one, the solution fails. If the value is zero, it implies the

[4] Problem set 10 are represented in Formulation I and problem set 25 are in Formulation II. Both formulations have been discussed in Section 4.3.2.

[5] i.e. 10.53, 10.55, 10.62, 10.64, 25.53, 25.55, 25.62 and 25.64

Table 5.7: PC configuration problems in the Comparison Phase.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	approx. density
10.3	14	10 – 40	1.01×10^{15}	14	2	14	1.1×10^{-1}
10.41	14	10 – 40	1.01×10^{15}	17	2	14	7.4×10^{-5}
10.53	14	10 – 40	1.01×10^{15}	16	2	14	-
10.55	14	10 – 40	1.01×10^{15}	16	2	14	-
10.62	14	10 – 40	1.01×10^{15}	16	2	14	no solution
10.64	14	10 – 40	1.01×10^{15}	16	2	14	no solution
25.3	59	2 – 40	6.72×10^{29}	31	1	14	-
25.43	59	2 – 40	6.72×10^{29}	35	1	14	-
25.47	59	2 – 40	6.72×10^{29}	39	1	14	-
25.50	59	2 – 40	6.72×10^{29}	45	1	14	-
25.53	59	2 – 40	6.72×10^{29}	33	1	14	-
25.55	59	2 – 40	6.72×10^{29}	33	1	14	-
25.62	59	2 – 40	6.72×10^{29}	33	1	14	no solution
25.64	59	2 – 40	6.72×10^{29}	33	1	14	no solution
20.3	32	2 – 40	9.81×10^{16}	20	1	12	-
21.3	36	2 – 40	5.89×10^{18}	20	1	12	-
22.3	46	2 – 40	3.62×10^{23}	23	1	12	-
23.3	51	2 – 40	1.16×10^{25}	23	1	14	-
24.3	58	2 – 40	3.36×10^{29}	31	1	14	-
25.3	59	2 – 40	6.72×10^{29}	31	1	14	-
26.3	70	2 – 40	1.14×10^{37}	31	1	15	-

Problem number beginning with 10 are from problem set 10, which are represented in Formulation I. Problem number beginning with 25 are from problem set 25, which are represented in Formulation II.

problem has been successfully solved. Otherwise, the result will be interpreted as an "acceptable" solution,[6] and that price constraints are the only constraints unsatisfied. The conflict count function weighing all constraints by their arity has no such benefit. Also, problems 62 and 64 are intentionally set to unsolvable with constraint '`UPPERprice(items, 500)`'. These are only used to test those algorithms using distance objective function. Different from problem 62, the search space of problem 64 is arranged in the order of the item price. Similarly, problem 53 and 55 are a pair of test problems. These two problems are solvable, but with a harder constraint '`UPPERprice(items, 750)`'. The search space of problem 53 is non-ordered and the search space of problem 55 is ordered.

[6] In other words, the problem is treated as optimization problem for the distance objective function.

5.2.2.3 Test problems for the all_diff phase

To observe the ability of the algorithms in handling n-ary constraints, we test the algorithms on n-queens problems for $n = 4, 5, 6, \ldots, 10$, and 15. The problem specifications are listed in Table 5.8.

Table 5.8: n-queens problems in the all_diff phase.

problem no.	var n	domain size	search space	constraints #	min. arity	max. arity	solution #	solution density
4-queens	4	4	256	13	2	4	2	7.81×10^{-3}
5-queens	5	5	3125	21	2	5	10	3.20×10^{-3}
6-queens	6	6	46656	31	2	6	4	8.57×10^{-5}
7-queens	7	7	823543	43	2	7	40	4.86×10^{-5}
8-queens	8	8	16777216	57	2	8	92	5.48×10^{-6}
9-queens	9	9	387420489	73	2	9	352	9.09×10^{-7}
10-queens	10	10	1.00×10^{10}	91	2	10	724	7.24×10^{-8}
15-queens	15	15	4.38×10^{17}	211	2	15	2279184	5.20×10^{-12}

5.2.3 Comparison measures

To evaluate the particle swarm algorithms and to answer the research questions, we will measure the effectiveness and efficiency of the algorithms. As particle swarms are closely related to evolutionary computing, we will use the measures in that field to evaluate the particle swarm algorithms. The discussion of Craenen et al. in [17] on comparing a set of evolutionary algorithms on solving binary CSPs provides a general reference.

5.2.3.1 Measuring effectiveness

PSOs are incomplete search algorithms and so cannot guarantee to find a solution if one exists. Therefore, to compare the effectiveness of the algorithms, we have two measures:

1. At first, we want to know whether the algorithms can solve a test problem and the probability for the algorithms to solve a problem if they cannot guarantee to solve it. For this, we can use the percentage of successful runs that the algorithms find solutions, i.e. the success rate (SR) as one of the measures.

2. Giving our constraint handling approaches, conflict counts and distance estimation,

the evaluation value (EV) of the objective functions equal to zero means a perfect solution has been found; otherwise, the higher the value, the worse the solution quality.[7] Since a particle swarm algorithm does not guarantee to converge on a perfect solution, the EV may not always be zero. The quality of a solution at termination[8] suggests the effectiveness of an algorithm. Considering that an algorithm will be tested multiple times, we will use a mean evaluation value (MEV) to estimate the effectiveness of the algorithms. A mean evaluation value comes from both the successful cases and the failures. Normally, an algorithm with a better success rate tends to have a lower MEV because whenever the algorithm solves a problem, its objective function yields zero. Without replacing the measure of MEV, we also want to examine odd cases where for example, an algorithm may have a good success rate on average but a high EV when it fails. For such cases, we divide the sum of the evaluation values of each algorithm over its number of failures and produce mean evaluation value on failures (FEV).

While examining the mean evaluation value and the mean evaluation value on failures, we must bear two issues in mind:

- The two objective functions measure solutions differently, so we cannot compare the values across the objective functions. We will separate the comparison into two groups: Continuous-Conflict, Discrete-Conflict and BCSP models as Group 1 for the algorithms that use the conflict count function, and Continuous-Distance and Discrete-Distance as Group 2 for the algorithms using the distance function.
- An evaluation value is only an estimation, not absolute. A potential solution with a lower evaluation value may not guarantee to find an actual solution faster. For those partial solutions with the same evaluation value, the quality may not be the same. So, we only use the mean evaluation value in supporting the results from evaluating the success rate.

5.2.3.2 Measuring efficiency

To evaluate efficiency, we have incorporated several variables into the test algorithms to keep track of program elapsed run time, the number of iterations and the number of consistency

[7]This is just a general rule, unless the objective function is able to provide perfect information of a potential solution.

[8]A PSO program terminates when either it finds a solution or it reaches a maximum number of iterations.

checks that the algorithms perform:

1. The elapsed run time (RT) that an algorithm takes to solve a problem or to exceed the iteration limit, is straightforward and generally provides an indication of the efficiency of an algorithm. After all, we want to know how long it would take to receive an answer from an algorithm. However, RT can be complex to analyze because of its dependency on the hardware and implementation. Without careful control, an elapsed time can be affected by the environment even more. Therefore, we need an auxiliary measure besides using run time. Similar to the mean evaluation value, we can compare the mean run time (MRT) of each algorithm over a number of test runs.

2. The number of iterations (IT) is an internal counter, which suggests how long a program takes. If a problem is not solved, the maximum number of iterations is recorded. Generally, the higher the number of iterations, the longer the run time. But, not all algorithms spend the same amount of time for each iteration. For instance, running different numbers of particles takes different amount of time for each iteration. One remedy is to multiple the number of iterations by the population of a swarm, and we then have the number of evaluations (ES) of an algorithm. Even so, some algorithms may spend more time on creating or evaluating a good potential solution, while others may quickly generate a so-so solution each time. In effect, measuring number of iterations or number of evaluations is not so good as measuring number of consistency checks discussed below.

3. The number of consistency checks (CC) is the number of verifying constraint violations of a current (potential) solution. It is usually considered as an **atomic operation** [17] for CSP algorithms. In other words, a consistency check is the most basic and critical operation in a CSP algorithm. It is performed by all the algorithms and generally consumes the most computational time. Many CSP algorithms traditionally use this measure to evaluate themselves against others. Although the number of consistency checks also ignores the setup time of potential solutions, in support of the comparison of the algorithm run time, it has the advantage over the number of iterations or number of evaluations for several reasons. Each algorithm may perform different numbers of consistency checks in each iteration and so take different amount of time to complete an iteration. Thus, the number of consistency checks can more accurately imply how much time an algorithm runs than the number of iterations can. Also, since it is

commonly used for evaluating CSP algorithms, we can compare not only among the PSO algorithms with it, but also between PSO algorithms and other CSP algorithms potentially.

In summary, we will evaluate the effectiveness of the particle swarm algorithms by their success rate (SR) with the support of the mean evaluation value (MEV), and compare their efficiency by the mean run time (MRT) with the support of the number of consistency checks (CC).

5.3 Experiments

Experiments on particle swarm algorithms are resource intensive; thus, reasonable limits have been imposed on the experiments. Program speed is a critical issue too; thus, much effort has been put into speeding up the programs as will be discussed in Section 5.3.3.

5.3.1 Runs

The experiment is set to collect 10 runs per algorithm per test problem. For each of the test problems, we randomly and independently generated 10 sets of initial solutions (or initial states) namely set A, B, C, ..., J. Each set contains 10 initial solutions. All the algorithms will begin with these 10 sets of initial solutions, one set per test run. Depending on the population of the swarm, an algorithm will take the first 3, 5 or 10 initial solutions to begin.

5.3.2 Experimental facilities

5.3.2.1 Computer hardware and software environment

Owing to the availability of the facilities and the nature of the experiments of different phases, we have employed different types of computers and software environment in each phase. In the Exploration phase, we wanted to determine what algorithms to study further and what parameter settings to work with, so using identical computers across the entire experiment was not critical. In order to complete this phase as soon as possible, we decided to use available computers in the labs. We only made sure that the same kind of computers and environments were used across the algorithms for one particular test problem. On the other hand, in order to evaluate as fairly as possible in the Comparison phase, we used a number of identical computers running the same operating system. In `all_diff` phase,

speed was not a major concern. Thus, all the algorithms were run on an Intel Pentium 4. The systems used in different phases of the experiment are listed in Table 5.9.

Table 5.9: The systems used in the three-phase experiment.

Phase	CPU	Memory MB	Number of PC	Operating system	Python version
Exploration	Intel P4, 3.0GHz	1024	5	Windows XP	2.4
	Intel P4, 3.0GHz	1024	1	Linux	2.3
	Intel P4, 2.8GHz	512	20–35	Windows XP	2.3
	Intel P4, 2.4GHz	512	1	Windows XP	2.4
	Intel P4, 2.4GHz	512	10–15	Windows XP	2.3
Comparison	Intel P4, 2.8GHz	512	20–35	Windows XP	2.3
`all_diff`	Intel P4, 2.4GHz	512	1	Windows XP	2.4

5.3.2.2 The accuracy of the experiment

When we discuss the evaluation measures in Section 5.2.3, we mentioned the accuracy of using real time run time, i.e. total elapsed time. Before we use it to evaluate the efficiency of the particle swarm algorithms (although our conclusion will be supported by the auxiliary measure—the number of consistency checks), we must understand what possible issues may affect the accuracy of the measure and the data.

Run time is straightforward; but it is affected by hardware, software environment and implementation besides the time a swarm actually needs to set up initial solutions, search, and evaluate solutions. For example, different CPUs give different performance, and different amounts of memory may support a program differently. Even with so-called "identical" machines, the CPUs may not perform exactly the same. Particularly, it is sometimes difficult to avoid every possible external process in Windows environment. If the run time of all the algorithm is mostly large, this effect will not be so significant. Otherwise, we should be aware the effect may be critical.

Another influence comes from the implementation of the algorithms such as the choice of language or particular techniques in use. For example, certain operations can be fast in Java but slow in Python, or vice versa. Depending on the required operations, there might be a case that algorithm A is faster than B if they are implemented in Java, but B is faster than A if they are in Python. One other issue also related to implementation is the memoization,

which will be discussed in the next section. In all, because of the problems just described, we should carefully consider and cannot rely on real-time as the sole measure of efficiency.

5.3.3 Programming issues

We choose Python as the implementation language. While Python is not generally the most efficient language available, it is a good choice for prototyping and experimenting with new algorithms.

In order to improve the speed, we used Python packages such as Pysco [79] and Python Numeric module [69] that approximately doubled the performance. Other optimizations were made based on the results of profiling the code, and using memoization[9] [9, 62]. However, it turns out that there is a downside to memoization. If the swarm keeps generating new assignments, the memory usage grows quickly. Eventually memory context switching may kick in, and the performance will considerably decrease. This happened sometimes during the experiment when the system ran out of memory. Thus, we should be aware of any abnormally huge and sudden changes in run time of an algorithm, which may be simply caused by running out of memory rather than by the algorithm itself.

5.4 Experimental results

We look at the experimental results from two aspects. One aspect is the particle swarm algorithms[10] of the three models. From the results, we can analyze and determine whether Schoofs and Naudts' algorithm [90] can be extended to solve n-ary CSPs and whether the traditional PSOs can be modified to solve n-ary CSPs.[11] Another aspect compares the results across the PSO models, from which we can determine if any of the three models is better or promising for solving n-ary CSPs.[12] To compare fairly among the PSO models, we only use the same types of algorithms from each model, i.e. the generic type, zigzag type, hop type and zigzagHop type algorithms to assess their performance. Several other algorithms based on *genericPSO* will be considered as individual algorithms. While analyzing the

[9]For the definition of memoization, see online material http://en.wikipedia.org/wiki/Memoization. An example in Python Cookbook at http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52201.

[10]See Table 4.1 for a list of algorithms as needed.

[11]These are for the first research question and the second research question.

[12]This corresponds to part of the second research question stated in Chapter 3.6.

algorithms, we mainly consider the overall performance across different parameter settings.

The resulting data came from both the Comparison phase and `all_diff` phase of the experiment.[13] The results of the `all_diff` phase are used to discuss the effectiveness of the particle swarm algorithms in handling n-ary constraints. The outcomes of the Comparison phase provide both the effectiveness and efficiency of the particle swarm algorithms, for which we look at the following measures:

- success rate (SR);
- mean evaluation value (MEV) (in support of success rate);
- run time (RT);
- number of consistency checks (CC) (in support of run time).

In this section, we present the results and briefly examine the outcomes. More specific discussion and analysis on the research questions will be in presented Section 5.5.

5.4.1 Effectiveness

5.4.1.1 The data from the Comparison phase

The overall success rate of the PSO models in Figure 5.1 shows that the conflict count objective function provides more help than the distance objective function in PC configuration problems.[14] If we verify the results of the mean evaluation values in Figure 5.2, the particle swarms using the distance function yield a slower growth as the complexity of the problems increases (the number of constraints of the problems increases). The lower success rate and flatter mean evaluation values suggest that the distance estimation does not provide sufficient information to distinguish the quality of solutions for the PC configuration problem and so prevents the swarm from improving solutions.

As groups of algorithms, the two Discrete models are slightly more effective than the Continuous models and the BCSP model on problems in Formulation I, but perform much worse than the Continuous models and the BCSP model on the problems in Formulation II[15]

[13] i.e. the experiments on the PC configuration problems and the n-queens problems, respectively

[14] Compared with problem set 25, problem set 10 are relatively simple. The difference between the two objective functions in problem set 10 is not obvious in the success rate.

[15] Problems in problem set 10 are formulated in Formulation I, and problem set 25 and problems 20.3 to 26.3 are in Formulation II.

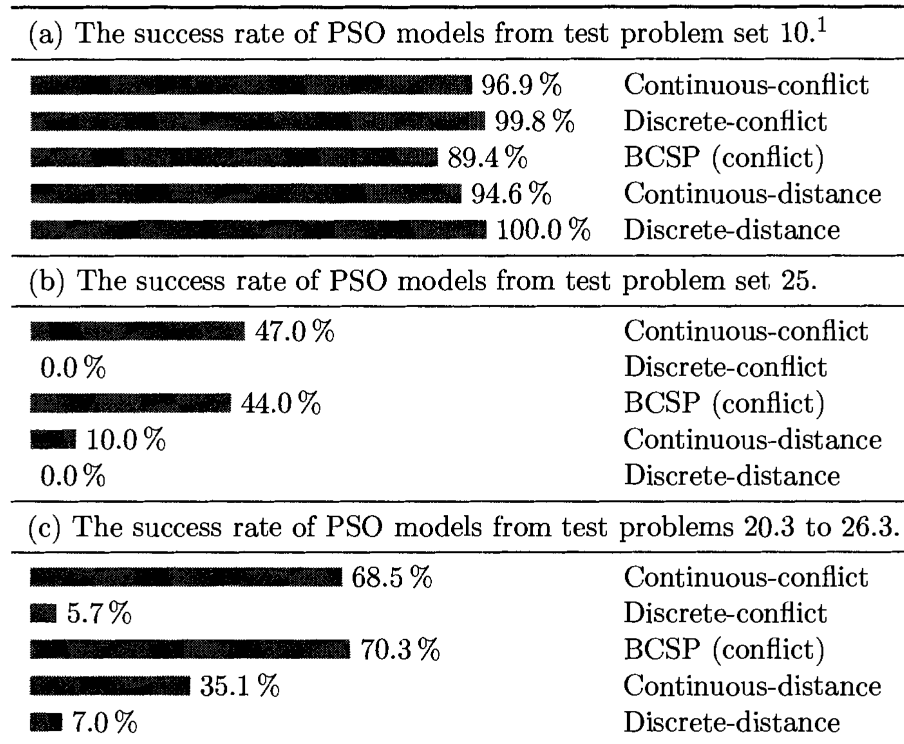


Figure 5.1: The success rate of PSO models from the Comparison phase.

The outcomes include all PC configuration problems from the Comparison phase, except for the result of problem set 10. The result of problem set 10 comes from only problem 10.3 and 10.41, which are simpler than problem 10.53, 55, 62 and 64. Problem 10.53, 55, 62 and 64 were not completed for *pop* = 10, so we do not take them into account. The partial results suggest that the success rates of problem set 10 are 54.4%, 59.9%, 52.0%, 57.9% and 65.0% respectively. Each model is assessed on the generic, zigzag, hop and zigzagHop type algorithms.

(see Figure 5.1 and Table 5.10). Formulation I and II represent different classes of problems. While the CSP domains in Formulation I are consecutive and most of the domains are consistent,[16] the domains of the test problems under Formulation II are not all consecutive and the domains across different CSP variables are much different. Being able to solve the test problems under Formulation I and not under Formulation II suggests that the Discrete models can solve the problems where the domains are consecutive and consistent, but do not perform well on the problems where the domains are non-consecutive or inconsistent.

[16]See Definition 2.1.1 for the definitions of consecutive domain and consistent domains. Also, refer to Section 4.2.1.2 for more discussions.

The Continuous-conflict model and the BCSP model have similar success rates on average, but the BCSP model yields lower mean evaluation values in Figure 5.2. Among the problems in problem set 25, the Continuous-conflict model has better success rate as the complexity of the problems increases. Both models have great difficulty with hard problems such as problems 25.53 and 25.55 (see Table 5.10 as needed).

None of the generic type algorithms[17] have impressive performances. They could not solve any problem in problem set 25 as shown in Table 5.11 and their mean evaluation values are relatively high compared with the other algorithms as shown in Figure 5.3. The generic BCSP algorithm (*bcspPSO*) is the worst among all the generic type algorithms for problems involving n-ary constraints such as problems 10.41, 20.3 to 26.3 and the problems in problem set 25.

For the performance of the proposed strategies, we focus on the "zigzag" movement and "no-hope and hop" strategy. Table 5.11 and Figure 5.3 show that these two strategies except for *genericZigzag*, greatly improve the *genericPSO*-conflict and *bcspPSO* algorithms for most of the test problems. Although *genericZigzag* does not improve *genericPSO*'s success rate by much, it does render lower mean evaluation values. The zigzagHop strategy combining the two strategies performs even better. *genericZigzagHop* and *bcspZigzagHop* are the best algorithms in this research, and *binaryZigzagHop*[18] is also the best among the discrete particle swarms. The "partner exchange" and "DFS" strategies on the other hand, do not contribute to the improvement very much.[19] Hence, the improvement that *genericHybrid*[20] has inflicted on the *genericPSO* algorithm mostly comes from the "no-hope and hop" strategy and the spawned particles.

Observing the data more closely, we can see how the swarm population and parameter *pop_rate*[21] affect the algorithms. Among the three population settings 3, 5 and 10, bigger

[17] i.e. those algorithms directly derived from the original PSOs including *genericPSO*, *binaryDiscrete*, *grayDiscrete* and *bcspPSO*

[18] Like other discrete algorithms, the performance of *binaryZigzagHop* is also restricted by the CSP domains.

[19] The "partner exchange" improves the mean evaluation value but it does not obviously improve the success rate; the "DFS" does not contribute to the improvement much in either the success rate or mean evaluation value.

[20] It integrates the "no-hope and hop", "parter exchange" and "DFS" strategies together, as well it will adaptively spawn more particles when no improvement has been done for too long.

[21] *pop_rate* in the "no-hope and hop" strategy defines the percentage of the population to perform the strategy. It applies to all the hop type and zigzagHop type algorithms and *genericHybrid* algorithm.

swarms are generally found to be more effective (see Figure 5.4 and Table C.4 as needed).[22] In the Comparison phase, we set *pop_rate* to 0.25, 0.5 and 0.75 for the hop type and zigzagHop type algorithms, and 0.2 for the *genericHybrid* algorithm.[23] Both the success rate and mean evaluation value versus *pop_rate* relations (in Figure 5.5 and Figure 5.6) suggest that with the higher *pop_rate* (i.e. more particles hop at the same time), the swarm performs more effectively in general.

We have mentioned that the improvement done by *genericHybrid* to the *genericPSO* algorithm mostly comes from the "no-hope and hop" strategy and the spawned particles. We can see that *genericHybrid*'s *pop_rate* is not so high as those of *genericHop* or *genericZigzagHop*, and so its success rate is not as good as those of the two, especially in problem 25.50. Also, *genericHybrid*'s success rates of different parameter settings on the same problem do not change much, which may suggest that those settings do not very effectively change the behaviour of the swarm (see Table C.4).

Comparing problem pairs (53 vs. 55) and (62 vs. 64), the price-ordered search space does not affect the algorithms very much in improving the quality of the search on these harder problems. To be more accurate, only *genericPSO*, *genericHop*, *genericZigzagHop*, *binaryDiscrete* and *binaryHop* are more sensitive to such an ordering and have a decreasing mean evaluation value in Figure 5.3.

In support of the success rate, the results of the mean evaluation value in Figure 5.3 and Figure 5.7 are generally consistent with those of the success rates as we have seen. While inspecting the mean evaluation value on failures, we do not find any inconsistent result. An additional observation from the mean evaluation values in Figure 5.7 is that most of the algorithms in the figure show obvious hikes at problem 25.50; those are more than likely caused by the 11-ary price constraints because the constraints contribute 1 for every dollar exceeding the budget to the distance objective function. For example, if some potential solution is $100 more than user budget, the solution would have at least EV = 100 when every other constraints are satisfied. Such a system implies user budget is absolutely critical. When an algorithm cannot resolve the price constraint, its mean evaluation value becomes high. The evaluation of the price constraint has been changed in problem 25.53, 55, 62 and 64 as described in Section 5.2.2.2, so the situation is mostly leveled out and the mean evaluation values are generally reduced.

[22] We only show several algorithms in Table C.4; the other algorithms share the similar observation.

[23] *genericHybrid* has many features to manipulate already, so we set only one *pop_rate* for the experiment.

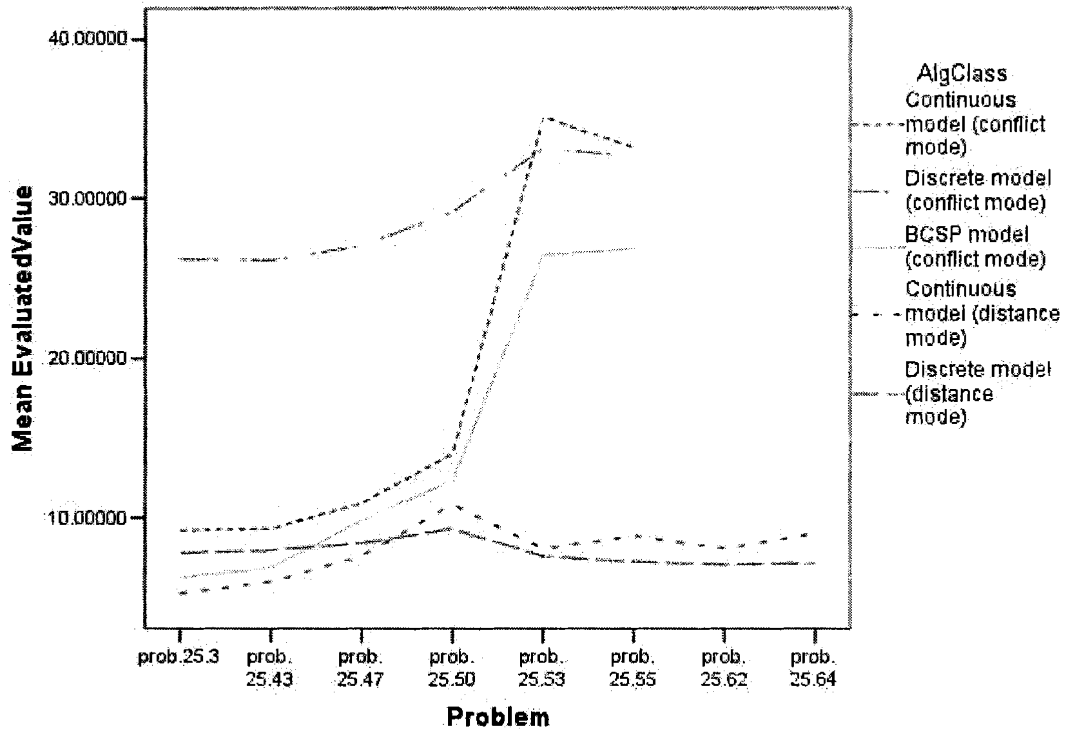


Figure 5.2: The mean evaluation value of PSO models from problem set 25.

1. See Figure C.2 for problem set 10 and problems 20.3~26.3.
2. All models have growing mean evaluation values (MEVs) while the complexity (the number of constraints) of the problems increases. The MEVs of the PSOs using the distance objective function grow at a slower rate.
3. The MEVs have an obvious escalation as the complexity of the problems increases. BCSP model is the best among the models using the conflict count function.
4. The price constraints in problems 25.53, 55, 62 and 64 of the PSOs using the distance function are measured in a much smaller scale, so we can see the MEVs going downward after problem 25.50 for both the Continuous-distance and Discrete-distance models.

Table 5.10: The success rate of PSO models from the Comparison phase.

The success rate of PSO models									
Class		Problem							
Model	Objective	10.3	10.41	10.53	10.55	10.62	10.64		
Continuous	conflict	100%	93.8%	0.6%	3.3%	-	-		
Discrete	conflict	100%	99.6%	0%	0%	-	-		
BCSP	conflict	100%	78.9%	0%	0.5%	-	-		
Continuous	distance	100%	89.2%	67.5%	90.8%	0%	0%		
Discrete	distance	100%	100%	71.3%	100%	0%	0%		
Model	Objective	25.3	25.43	25.47	25.50	25.53	25.55	25.62	25.64
Continuous	conflict	72.1%	70.8%	67.9%	53.8%	4.2%	13.3%	-	-
Discrete	conflict	0%	0%	0%	0%	0%	0%	-	-
BCSP	conflict	73.5%	73.0%	60.6%	51.9%	2.4%	2.6%	-	-
Continuous	distance	28.8%	27.9%	19.2%	3.8%	0%	0%	0%	0.4%
Discrete	distance	0%	0%	0%	0%	0%	0%	0%	0%
Model	Objective	20.3	21.3	22.3	23.3	24.3	25.3	26.3	
Continuous	conflict	77.1%	77.1%	72.9%	72.1%	72.1%	72.1%	36.3%	
Discrete	conflict	28.8%	11.3%	0%	0%	0%	0%	0%	
BCSP	conflict	78.9%	78.1%	75.4%	75.6%	72.8%	73.5%	38.1%	
Continuous	distance	51.3%	51.3%	38.3%	37.1%	27.5%	28.8%	11.3%	
Discrete	distance	37.1%	11.7%	0%	0%	0%	0%	0%	

1. The outcomes are the individual PC configuration problems from the Comparison phase.
2. Each model includes the generic, zigzag, hop and zigzagHop type algorithms.
3. The Discrete model performs effectively in Formulation I, but can solve only few test problems in Formulation II.
4. The Continuous-conflict model and BCSP model perform competitively; but, the Continuous model handles harder problems (as problems 10.41, 25.47, 25.50 and up) slightly better than the BCSP model, except for problem 26.3. However, none of the models have acceptable performance on problems 25.53 and 25.55.
5. Problems 10.53, 55, 62 and 64 were not completed for *pop* = 10 so the results shown are only partial. Since swarms with *pop* = 3 or 5 are not usually as effective as those with *pop* = 10, the partial results appear to be really low. We might expect better result if the experiment had been done.

Table 5.11: The success rate of PSO algorithms from the Comparison phase.

PSO Algorithm	Continuous conflict			Discrete conflict			BCSP conflict			Continuous distance			Discrete distance		
	I	II	III	I	II	III	I	II	III	I	II	III	I	II	III
genericPSO	85%	0%	2%	-	-	-	-	-	-	90%	0%	11%	-	-	-
genericZigzag	90%	0%	3%	-	-	-	-	-	-	92%	.4%	13%	-	-	-
genericHop	100%	59%	90%	-	-	-	-	-	-	100%	26%	74%	-	-	-
genericZigzagHop	100%	67%	91%	-	-	-	-	-	-	92%	.3%	12%	-	-	-
genericExchange	100%	.2%	-	-	-	-	-	-	-	98%	.2%	-	-	-	-
zigzagExchange	99%	0%	-	-	-	-	-	-	-	98%	.8%	-	-	-	-
zigzagDFS	87%	0%	-	-	-	-	-	-	-	91%	.3%	-	-	-	-
genericHybrid	100%	47%	86%	-	-	-	-	-	-	100%	32%	86%	-	-	-
binaryDiscrete	-	-	-	98%	0%	2%	-	-	-	-	-	-	100%	0%	5%
binaryZigzag	-	-	-	100%	0%	1%	-	-	-	-	-	-	100%	0%	2%
binaryHop	-	-	-	100%	0%	2%	-	-	-	-	-	-	100%	0%	3%
binaryZigzagHop	-	-	-	100%	0%	12%	-	-	-	-	-	-	100%	0%	13%
grayDiscrete	-	-	-	97%	0%	0%	-	-	-	-	-	-	100%	0%	.5%
grayZigzag	-	-	-	98%	0%	0%	-	-	-	-	-	-	98%	0%	0%
grayHop	-	-	-	100%	0%	.2%	-	-	-	-	-	-	100%	0%	.3%
bcspPSO	-	-	-	-	-	-	52%	0%	.5%	-	-	-	-	-	-
bcspZigzag	-	-	-	-	-	-	86%	31%	61%	-	-	-	-	-	-
bcspHop	-	-	-	-	-	-	100%	53%	86%	-	-	-	-	-	-
bcspZigzagHop	-	-	-	-	-	-	99%	64%	95%	-	-	-	-	-	-

The outcomes include PC configuration problems from the Comparison phase. Column I's are the outcomes of problems 10.3 and 10.41 only (for the partial results of problem set 10, see Table C.3 as needed); Column II's are the outcomes of problem set 25; and Column III's are the outcomes of problems 20.3 to 26.3.

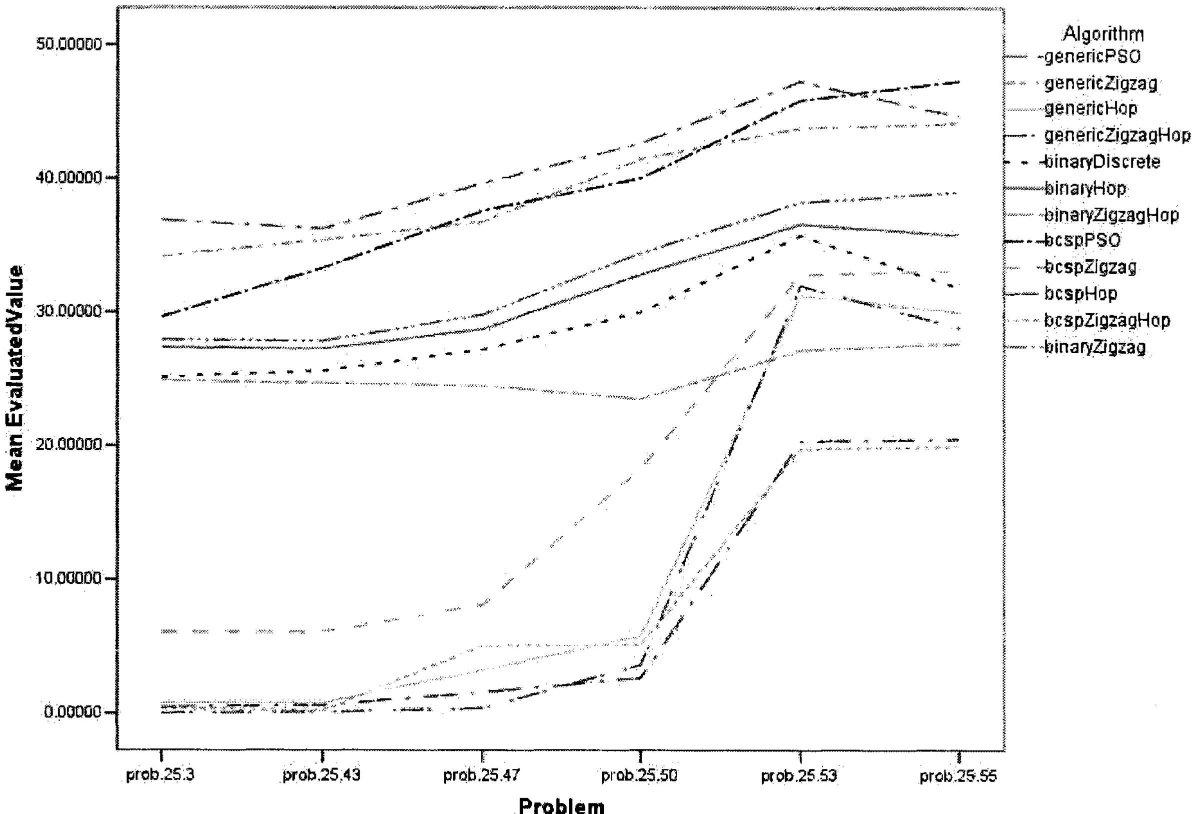


Figure 5.3: The mean evaluation value of PSO algorithms from problem set 25.

1. The outcomes present the individual PC configuration problem set 25 from the Comparison phase. These algorithms use the conflict count function.
2. We can roughly divide the algorithms into four groups in this figure. The *bcspPSO* algorithm along with *genericPSO* and *genericZigzag* stay at the top, *genericZigzagHop*, *genericHop*, *bcspZigzagHop* and *bcspHop* are at the bottom, *bcspZigzag* raising gradually is alone slightly above the bottom group and the rest of discrete algorithms are in the middle above *bcspZigzag*. The MEVs of the lower group start smoothly and suddenly increase at problem 25.53.
3. One special case is *binaryZigzagHop*, which appears a modest decrease between problem 25.3 to 25.50 and then grows again at problem 25.53 and 55. This implies that *binaryZigzagHop* performs and scales better than the other discrete algorithms. Given enough time, it may potentially solve the problems.

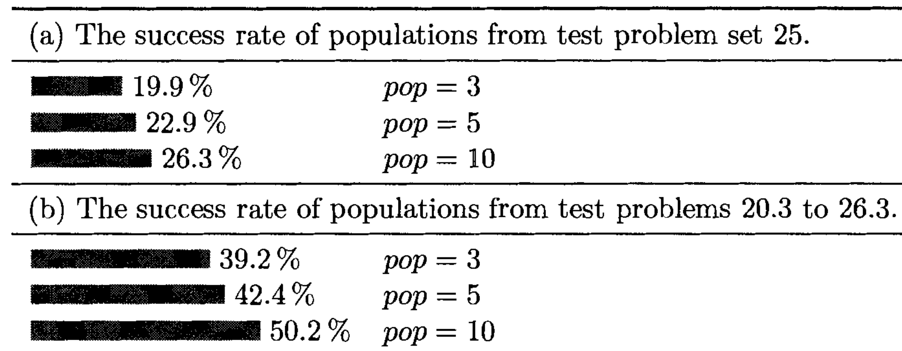


Figure 5.4: The success rate of populations from the Comparison phase.

1. The outcomes include all PC configuration problems in Formulation II from the Comparison phase. Each model includes the generic, zigzag, hop and zigzagHop type algorithms.
2. The result of problem set 10 is incomplete and not shown. If we consider only the completed problems 10.3 and 10.41, the success rates are 92.8%, 95.1% and 96.5% with respect to *pop* = 3, 5, and 10. The partial results of the problems in problem set 10, the success rates of problem set 10 become 49.3%, 52.7% and 75.3% respectively.

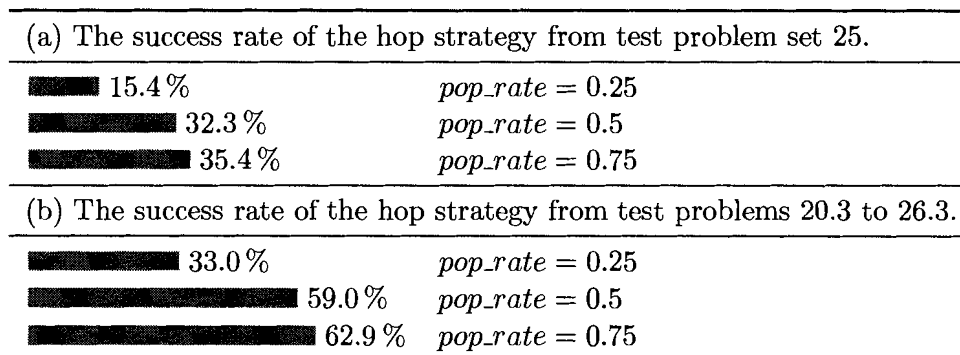


Figure 5.5: The success rate of *pop_rate*: hop and zigzagHop algorithms from the Comparison phase.

1. The outcomes include all PC configuration problems under Formulation II from the Comparison phase.
2. The result of problem set 10 is incomplete and not shown. If we consider only the completed problem 10.3 and 10.41, the success rates are 99.4%, 98.8% and 99.5% with respect to *pop_rate* = 3, 5, and 10. The partial results of the problems in problem set 10, the success rates of problem set 10 become 63.3%, 61.1% and 61.1% respectively.

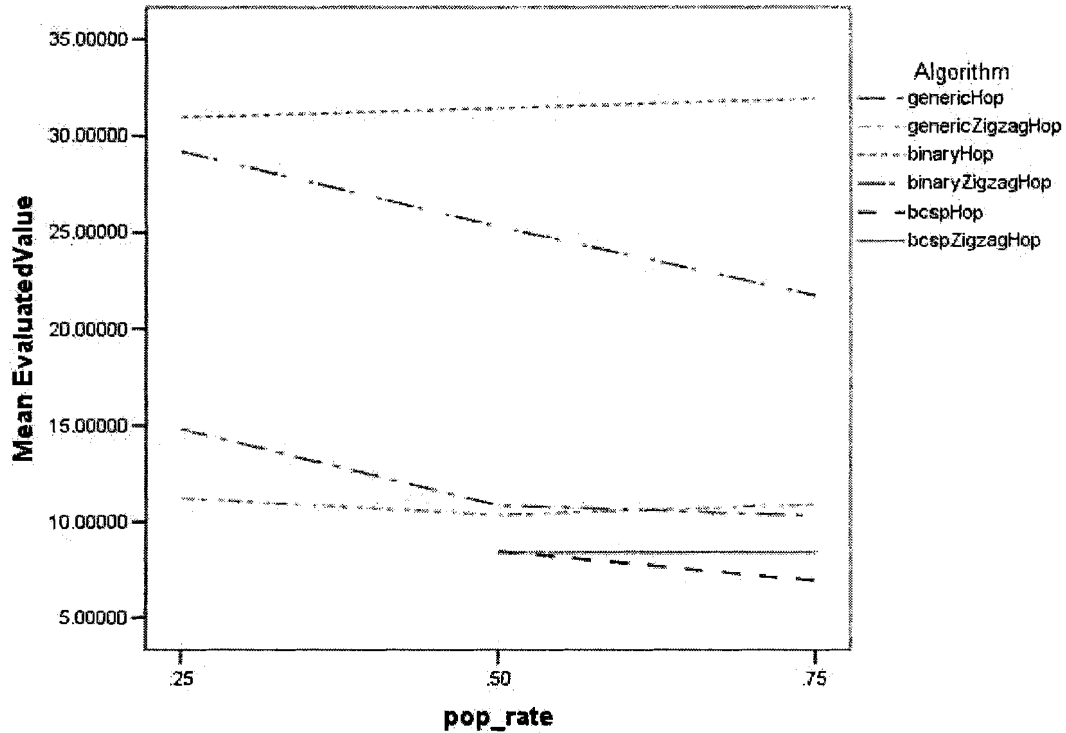


Figure 5.6: The mean evaluation value of *pop_rate* from problem set 25.

1. The outcomes include PC configuration problems in problem set 25 from the Comparison phase. The algorithms are the hop and zigzagHop algorithms using the conflict count function. See Figure C.1 for the algorithms using the distance function as needed.
2. Generally, the higher the *pop_rate*, the lower the mean evaluation value. Some changes between *pop_rate* $= 0.5$ and 0.75 are not obvious because *pop_rate* $= 0.5$ renders relatively high success rate on the test problems and *pop_rate* $= 0.75$ has only limited improvement, and the mean evaluation value of *binaryHop* slightly increases when *pop_rate* grows.

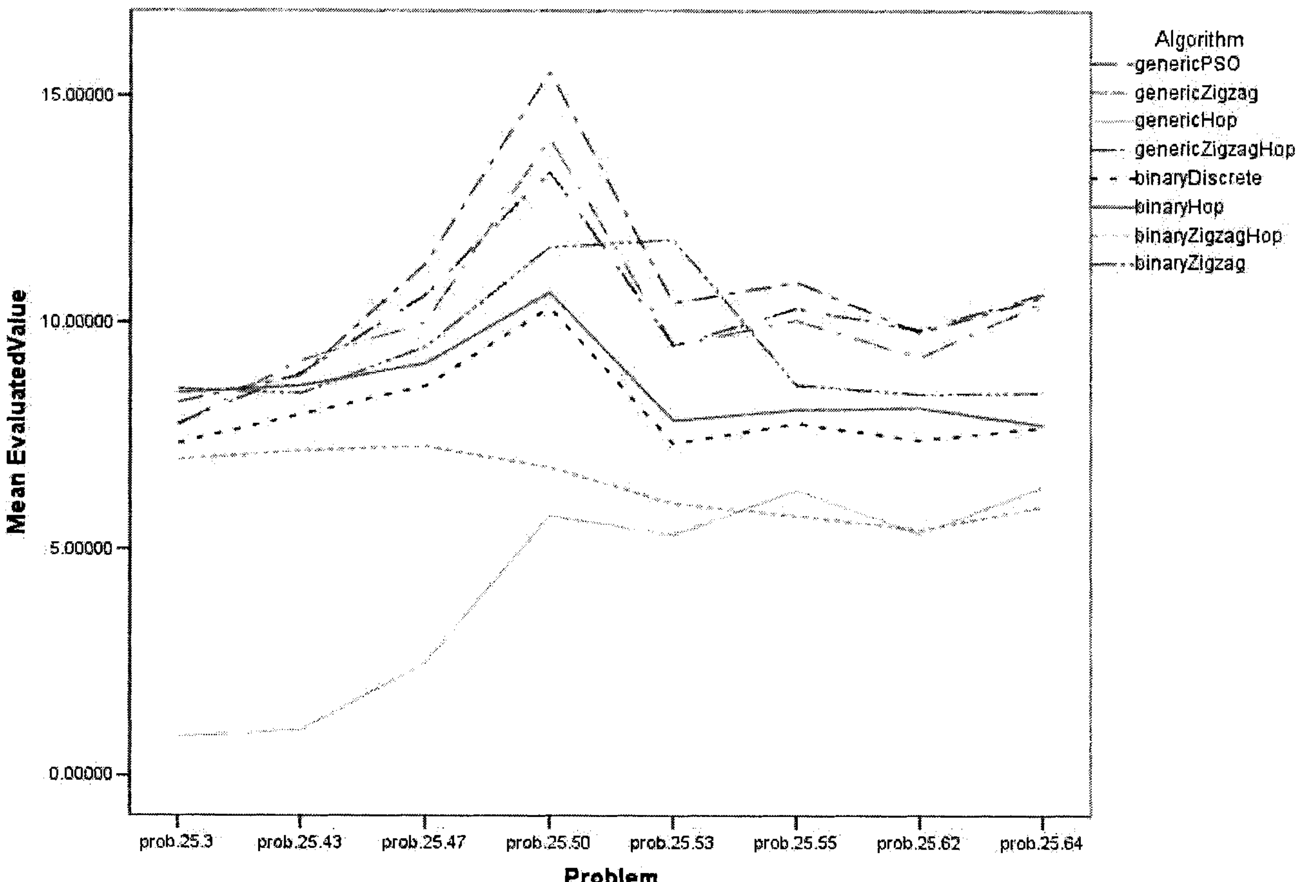


Figure 5.7: The mean evaluation value of PSO algorithms from the Comparison phase.

1. The outcomes present the individual PC configuration problems in problem set 25 from the Comparison phase. These algorithms use the distance function.
2. Only *genericHop*'s and *binaryZigzagHop*'s MEVs are in the middle of the figure, and the others are above them. Like *binaryZigzagHop*-conflict in Figure 5.3, *binaryZigzagHop*-distance here appears a little bit of decrease as the complexity of the problems increases.

5.4.1.2 From the all_diff phase – the n-queens experiments

The experimental results of the Exploration and Comparison phases show that the particle swarm algorithms have difficulty in dealing with n-ary constraints for large n. To observe the ability of the particle swarm algorithms to deal with n-ary constraints, we examine the results from the `all_diff` phase. The arity of the `all_diff` constraints equals to the number of queens.

The success rate of the PSO models and particle swarm algorithms are shown in Figure 5.8 and Table 5.12 respectively, from which we have several observations. First, none of the algorithms can solve 15-queens problem. Although most of the algorithms have their success rates above zero percent on 10-queens, only *bcspZigzagHop* is most promising. Overall, the zigzagHop type algorithms perform best among the algorithms studied.

Second, the Discrete models perform much better in this phase than they did in the Comparison phase on PC configuration problems under Formulation II. The domains of n-queens problems are consecutive and consistent,[24] and the fact that the discrete algorithms can solve n-queens problems confirms our previous observation. That is, the Discrete models can solve the problems where the domains are consecutive and consistent, but do not perform well on the problems where the domains are not consecutive or not consistent.

Third, the distance objective function works better with n-queens problems than it does with PC configuration problems. We have seen that the distance objective function does not provide sufficient estimation for good list or bad list constraints and price constraints. An n-queens problem contains only several arithmetic relation constraints and an `all_diff` constraint, for which the distance objective function provides better estimation and performs competitively to the conflict count objective function.

Table 5.12: The SR of PSO algorithms from the all_diff phase.

The success rate of PSO algorithms on n-queens problems								
Algorithm	4-q	5-q	6-q	7-q	8-q	9-q	10-q	15-q
Continuous-conflict	91.3%	93.8%	84.6%	89.2%	78.3%	55.8%	10.4%	0%
genericPSO	66.7%	66.7%	40%	56.7%	23.3%	0%	0%	0%
genericZigzag	63.3%	83.3%	46.7%	60%	50%	26.7%	0%	0%

[24]See Definition 2.1.1 for definition as needed.

The success rate of PSO algorithms on n-queens problems								
Algorithm	4-q	5-q	6-q	7-q	8-q	9-q	10-q	15-q
genericHop	100%	100%	96.7%	98.9%	85.6%	60%	8.9%	0%
genericZigzagHop	100%	100%	100%	100%	98.9%	80%	18.9%	0%
genericExchange	87.5%	95%	81.3%	87.5%	63.8%	22.5%	0%	0%
zigzagExchange	86.3%	93.8%	86.3%	91.3%	76.3%	36.3%	5%	0%
zigzagDFS	70.8%	84.2%	41.7%	60.8%	30%	8.3%	0%	0%
genericHybrid	100%	100%	100%	100%	77.5%	28.8%	1.3%	0%
Discrete-conflict	100%	100%	96.3%	100%	87.1%	50.4%	20%	0%
binaryDiscrete	100%	100%	76.7%	100%	73.3%	10%	3.3%	0%
binaryZigzag	100%	100%	93.3%	100%	90%	20%	6.7%	0%
binaryHop	100%	100%	100%	100%	80%	46.7%	16.7%	0%
binaryZigzagHop	100%	100%	100%	100%	97.8%	77.8%	33.3%	0%
grayDiscrete	100%	100%	76.7%	93.3%	56.7%	53.3%	6.7%	0%
grayZigzag	100%	100%	96.7%	93.3%	76.7%	56.7%	6.7%	0%
grayHop	100%	100%	98.9%	98.9%	61.1%	33.3%	4.4%	0%
BCSP-conflict	88.0%	85.7%	75.4%	76.5%	63.9%	58.7%	35.7%	0.6%
bcspPSO	31.1%	24.4%	1.1%	4.4%	0%	0%	0%	0%
bcspZigzag	96.7%	90%	67.8%	70%	67.8%	62.2%	26.7%	0%
bcspHop	100%	100%	91.7%	92.2%	58.3%	46.1%	17.2%	0%
bcspZigzagHop	100%	100%	100%	100%	99.4%	98.9%	76.7%	1.7%
Continuous-distance	88.4%	95%	77.5%	82.9%	56.3%	33.3%	9.6%	0%
genericPSO	73.3%	80%	60%	76.7%	40%	30%	13.3%	0%
genericZigzag	66.7%	80%	46.7%	66.7%	60%	60%	20%	0%
genericHop	100%	100%	87.8%	88.9%	56.7%	30%	6.7%	0%
genericZigzagHop	88.9%	100%	83.3%	84.4%	60%	35.6%	7.8%	0%
zigzagDFS	71.7%	85.8%	47.5%	71.7%	52.5%	22.5%	4.2%	0%
genericHybrid	100%	100%	100%	100%	81.3%	47.5%	8.8%	0%
Discrete-distance	100%	100%	96.7%	99.2%	93.8%	66.3%	26.3%	0.8%
binaryDiscrete	100%	100%	76.7%	96.7%	90%	26.7%	10%	3.3%
binaryZigzag	100%	100%	96.7%	96.7%	93.3%	46.7%	13.3%	3.3%
binaryHop	100%	100%	100%	100%	90%	62.2%	27.8%	0%
binaryZigzagHop	100%	100%	100%	100%	98.9%	90%	34.4%	0%
grayDiscrete	100%	100%	80%	96.7%	73.3%	66.7%	43.3%	3.3%
grayZigzag	100%	100%	96.7%	100%	86.7%	80%	50%	0%

The success rate of PSO algorithms on n-queens problems								
Algorithm	4-q	5-q	6-q	7-q	8-q	9-q	10-q	15-q
grayHop	100%	100%	97.8%	97.8%	70%	56.7%	20%	0%

5.4.2 Efficiency

We have discussed the complication of using (elapsed) real run time to evaluate the efficiency of the algorithms in Section 5.2.3.2 and 5.3.2.2. While investigating the run time of the particle swarm algorithms, we did see some exceptionally high run time, irregular data and suspicious huge hikes in various cases such as in Figure 5.9, Figure 5.10 and Figure 5.11. One reason for these high run time and irregular hikes in the graphs to occur could be the actual performance of the algorithms under certain parameter settings, and some algorithms really need higher run time. In such cases, the algorithms have more consistently high run time across a number of test runs.

The memoizer explained in Section 5.3.3 running out of memory could be another cause of the irregular high run time and the sudden hikes in the graphs. These cases create extremely huge mean run time error bars[25] and can usually be detected from the raw data. One example is the hike in Figure 5.9. Examining both Figure 5.9 and Figure 5.10, we can see that *binaryZigzagHop* as a hybrid of a zigzag type algorithm and a hop type algorithm is suspicious. The program run time increases irregularly while the number of consistency checks does not increase that much. Although hop type algorithms have higher run time per iteration, zigzag type algorithms generally run fast. From the raw data summarized in Table 5.13, we find that *binaryZigzagHop* could not solve problem 25.50 and ran for 50000 iterations each time. The Python object garbage collector did not take place between each test run, so the memory usage exceeded what we expected at the end of test runs. The huge amount of memory usage caused the system running out of memory. If we remove the last exceptional runs from the data, the hike at problem 25.50 disappears.

Looking at the overall run time of the PSO models, we can see that the BCSP model and the Continuous-conflict model are much more efficient than the two Discrete models. The

[25]We generated error bars while computing the mean of run time of the algorithms with confidence interval level set to 95%.

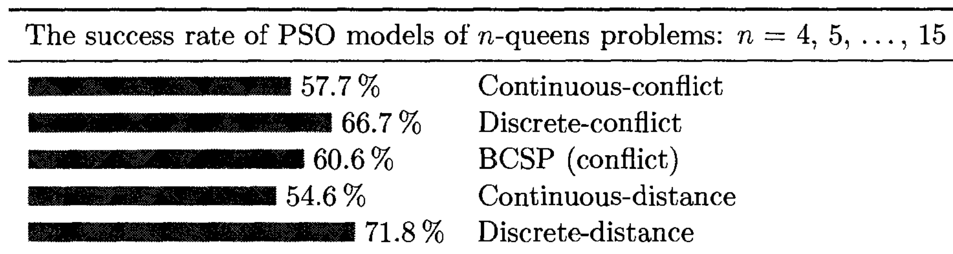


Figure 5.8: The success rate of PSO models from the all_diff phase.

Table 5.13: The average run time and the number of consistency checks of *binaryZigzagHop*-distance.

pop	*pop_rate*	avg. CC	avg. RT
3	0.25	$\approx 3.0 \times 10^5$	85.0
3	0.5	$\approx 3.6 \times 10^5$	102.6
3	0.75	$\approx 3.85 \times 10^5$	118.3
5	0.25	$\approx 5.0 \times 10^5$	134.1
5	0.5	$\approx 5.9 \times 10^5$	165.0
5	0.75	$\approx 6.5 \times 10^5$	197.6
10	0.25	$\approx 10.7 \times 10^5$	281.1
10	0.5	$\approx 13.2 \times 10^5$	384.6
10	0.75	$\approx 14.4 \times 10^5$	39244.1

On problem 25.50 for 50000 iterations.

Continuous-distance model performs as efficiently as the BCSP model and the Continuous-conflict model on the test problems in Formulation II, but it is much more expensive on problems 10.3 and 10.41. Verifying the results with the number of consistency checks, the BCSP model has the best on average, closely followed by the two Continuous models for the problems in Formulation II. The Continuous-conflict model however has the highest number of consistency checks for the problems in Formulation I.

Examining Figure 5.9, Figure 5.10, Figure C.4 and the average run time per iteration of the particle swarm algorithms, we can see some large mean run time at problems 10.41 and 25.50. Running out of memory is the major reason for the huge hike at problem 25.50.[26]

[26] See Figure 5.9 for detail as needed.

Investigating the high run time at problem 10.41 requires more care. The numbers of consistency checks of the algorithms are not exceptionally high at problem 10.41 (see Figure C.11 as needed). The algorithms using the distance objective function spend significant amount of time evaluating the potential solutions in each iteration for problem 10.41, which contributes the considerably high run time at the problem.[27] For the issues between problems 25.50 and 10.41, we have two observations.

First, the price constraint is more expensive to evaluate in problem 10.41 than it is in problem 25.50 because the arity of the constraint is higher in problem 10.41. Also, the memoizer potentially works better in problem 25.50 than in problem 10.41. Problem 10.41 has 14 variables and all the variables are involved in the price constraint; if any of the 14 variables changes its assignment, the assignment to the price constraint is changed. If the newly generated assignment is not in the memoizer, the evaluation computation must be done. On the other hand, problem 25.50 has 59 variables and only 11 variables are involved in the price constraint. So the chances to change a value from one of the 11 variables in the price constraint in an iteration is relatively lower than the 14 variables in problem 10.41. In turn, the algorithm may have a better chance to take advantage of the memoizer and so the evaluation is faster. This is especially true in zigzag type algorithms since they deal with one variable at a time.

Second, the average run time per iteration of the PSO algorithms indicates that the conflict count function evaluates a potential solution faster than the distance function does, particularly on `GOODLIST` and `BADLIST` constraints. The conflict count function only needs to check whether an assignment exists in the good list/bad list or not. Besides checking for existence, the distance function requires additional computation to estimate the quality of the assignment. The connection constraints[28] in problem set 10 are all good list and bad list constraints whereas those in problem set 25 are mostly in arithmetic forms. Even with a few `GOODLIST` and `BADLIST` component constraints[29] in problem set 25, the lengths of the lists are much shorter than those in problem set 10. Therefore, it is more expensive for the distance function to estimate the quality of the assignment for the problems under Formulation I.

[27]This is not so significant to the algorithms using the conflict count function.

[28]Enforcing the compatibility between components, the length of these constraints can be up to some cross-product of the n components, typically 2 or 3 components at least.

[29]Defining components and their specifications, the length of these constraints is the number of the component items.

As for the efficiency of the individual particle swarm algorithms, the zigzag type and zigzagHop type algorithms generally run faster and have lower number of consistency checks as shown in Figure 5.10, Figure 5.11, Figure 5.13, and Figure 5.14. Even with a higher iteration limit,[30] the zigzag type algorithms tend to maintain stable run time and number of consistency checks. The algorithms involving "hop" strategy on the other hand, tend to vary in run time and be more sensitive to the difficulty of the problems. Among the algorithms using the conflict count function, *genericZigzagHop* and *bcspZigzagHop* are the most efficient algorithms besides *zigzagDFS* and *genericZigzag*. Algorithm *genericZigzagHop* scales better and exhibits consistency with respect to its speed from problem 25.3 through 25.50. Algorithm *bcspZigzagHop* starts slightly better than *genericZigzagHop*. *genericZigzagHop* becomes more efficient than *bcspZigzagHop* as the complexity of the problems increases. We can conclude similarly from examining the number of consistency checks in Figure 5.13 and Figure 5.14. For large problems such as problem 26.3, *bcspZigzagHop* performs better than *genericZigzagHop*.

Similar to what we found in Section 5.4.1, we also noticed that the discrete algorithms perform competitively to the other algorithms (using the same strategies) on the problems in Formulation I,[31] but badly on problems in Formulation II. Especially, *binaryDiscrete* and *binaryHop* are the worst among all the algorithms in Figure 5.10, Figure 5.11, Figure 5.13, and Figure 5.14. Algorithm *binaryZigzagHop* runs rather fast compared with other discrete algorithms shown in Figure 5.10 and Figure 5.11, but it is still slower than the other algorithms.

Algorithm *bcspPSO* cannot solve any problems in problem set 25, but its run time remains relatively fast among the BCSP swarms. *bcspZigzag* is relatively slow among the BCSP algorithms. *bcspZigzagHop* starts as the fastest algorithm, but its run time gradually grows as the complexity of the problems increases. We will discuss more about these algorithms in Section 5.5.1.1.

As for parameter settings, we investigate the swarm population *pop* and the percentage of the population *pop_rate* to execute 'hop' strategy. The more the number of particles, the more the computational cycles translated into updating particles' velocities and positions,

[30] The zigzag type and zigzagHop type algorithms can run up to 50000 iterations versus the other algorithms only run up to 20000 iterations.

[31] Figure C.4, Figure C.6, Figure C.9 and Figure C.11 also show the low numbers of consistency checks of those discrete algorithms on problems in problem set 10.

and propagating the knowledge to the neighbours. If a problem is too hard, a large swarm naturally requires more time to complete and performs many more consistency checks. But if a large swarm can solve a problem more effectively than a small swarm, the total run time can be shorter and the number of consistency checks can be lower. We can generally see that the change of the mean run time versus population tend to be more insensitive to those effective algorithms although we still find that the bigger the swarm, the higher the mean run time.[32] For those ineffective algorithms such as *binaryDiscrete*, *binaryHop*, *grayDiscrete* and *grayHop*, more number of particles definitely implies more time to terminate.

Similar to *pop*, higher *pop_rate* means more particles to perform 'hop' strategy at the same time and each 'hop' takes time to determine which variables to fix. Thus, an algorithm with higher *pop_rate* generally requires more time and be less efficient to complete the search. However, an algorithm with a higher *pop_rate* may be more effective and so more efficient to solve a problem. These two factors offset the effects of each other and so *pop_rate* does not change the efficiency of the algorithms very much. Some algorithms such as *genericHop* and *binaryHop* can even have better efficiency with higher *pop_rate* as shown in Figure 5.15 and Figure 5.16.

[32]See Figure C.13 and Figure C.14 as needed.

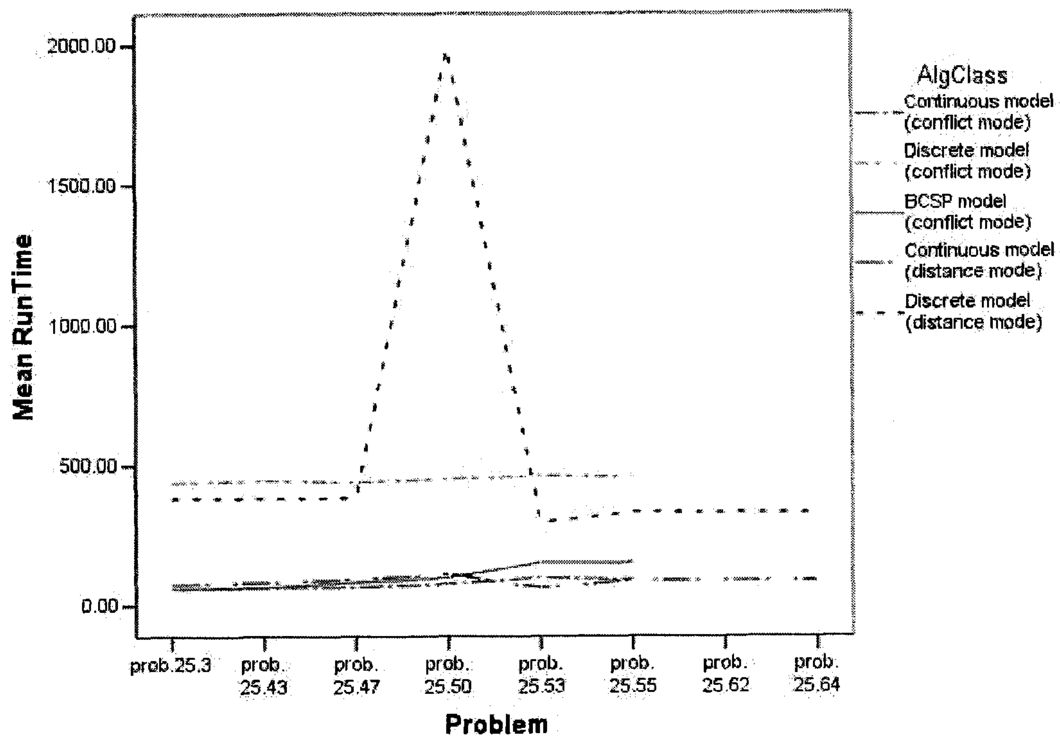


Figure 5.9: The mean run time of PSO models from problem set 25.

1. The outcomes present the individual PC configuration problems in problem set 25 from the Comparison phase. See Figure C.3(a) and Figure C.3(b) for problem set 10 and problems 20.3 to 26.3, respectively.
2. The problems in problem set 25 are hard for both Discrete models. Because these algorithms cannot solve the problems, they run to the iteration limit and then quit. Hence, we can see their mean run times are relatively flat at a level from problems 25.3 to 25.55.

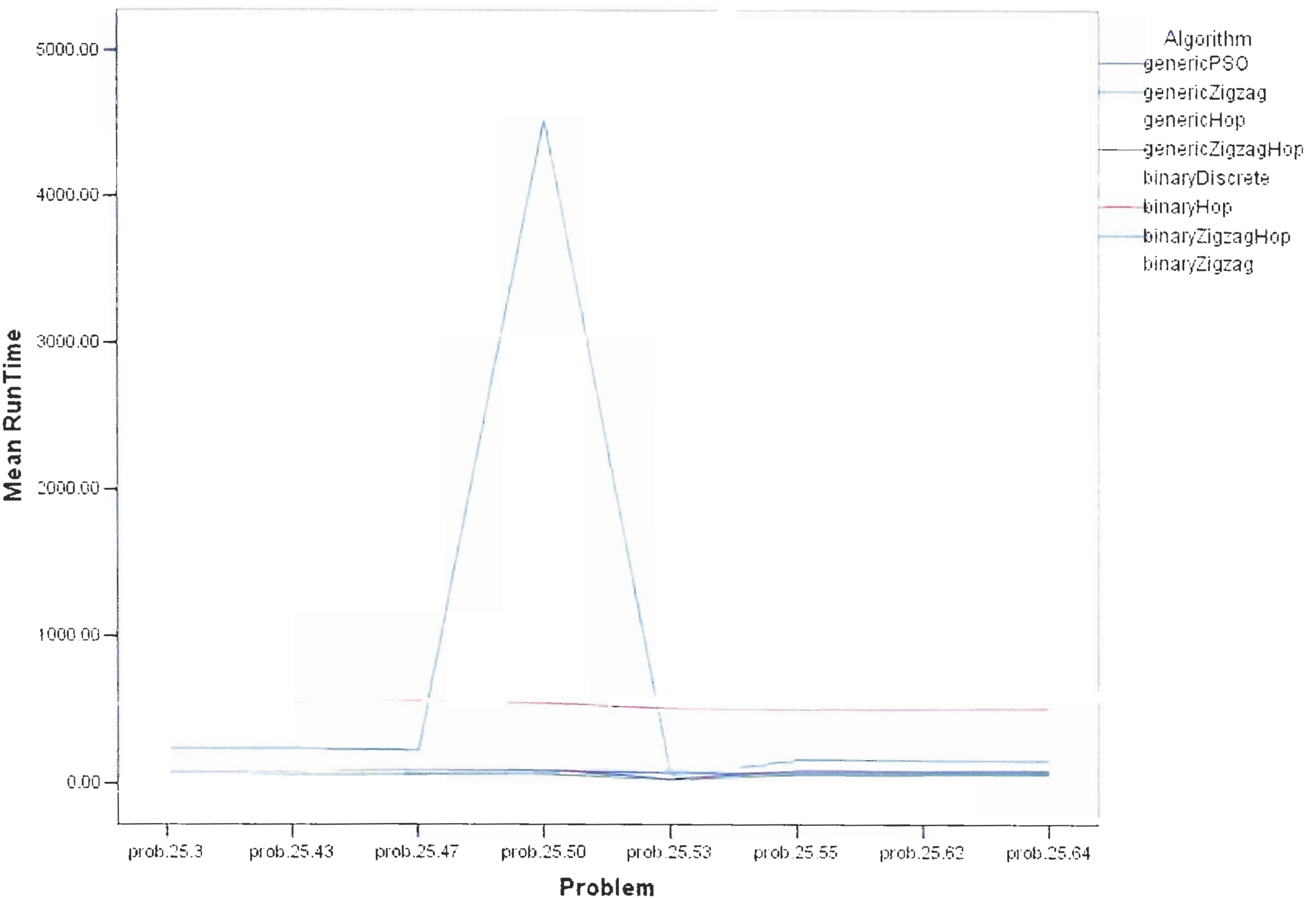


Figure 5.10: The mean run time of PSO algorithms from problem set 25.

The outcomes include PC configuration problems, problem set 25 from the Comparison phase. These algorithms use the distance objective function. See Figure C.4 and Figure C.5 for problem set 10 and problems 20.3~26.3, respectively. The mean run times of problem set 25 appears to be very close to one another among the algorithms. One explanation is the scale on y-axis. Changes within 10 or 100 seconds may be hard to observe. The hike at problem 25.50 has been explained in Section 5.4.2.

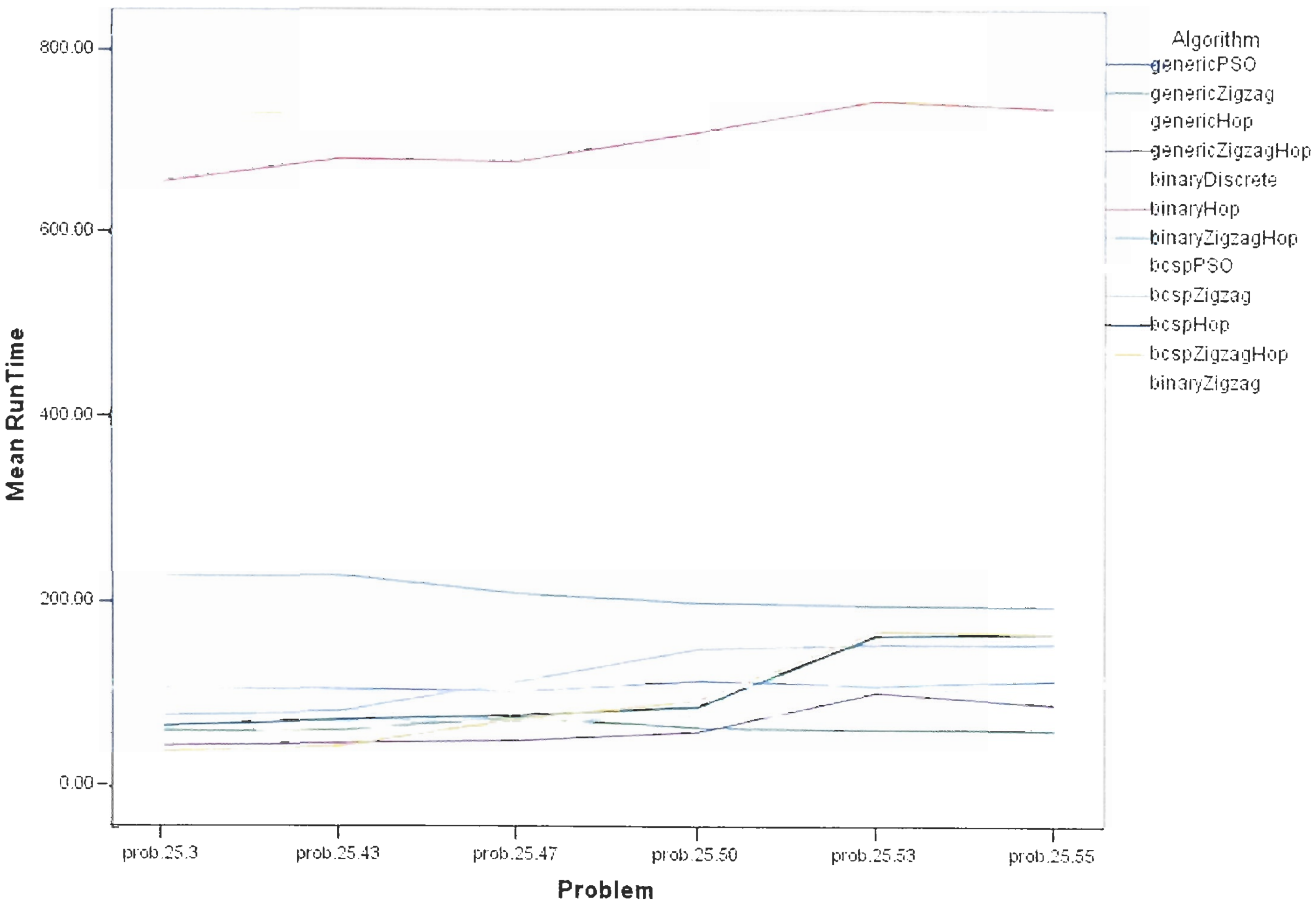


Figure 5.11: The mean run time of PSO algorithms from problem set 25.

The outcomes include PC configuration problem set 25 from the Comparison phase. These algorithms use the conflict count function. See Figure C.6 and Figure C.7 for problem set 10 and problems 20.3~26.3, respectively. A few algorithms such as *bcspPSO*, *genericPSO*, *genericZigzag* and *binaryZigzag* cannot solve problems in problem set 25, but their run time retain stable.

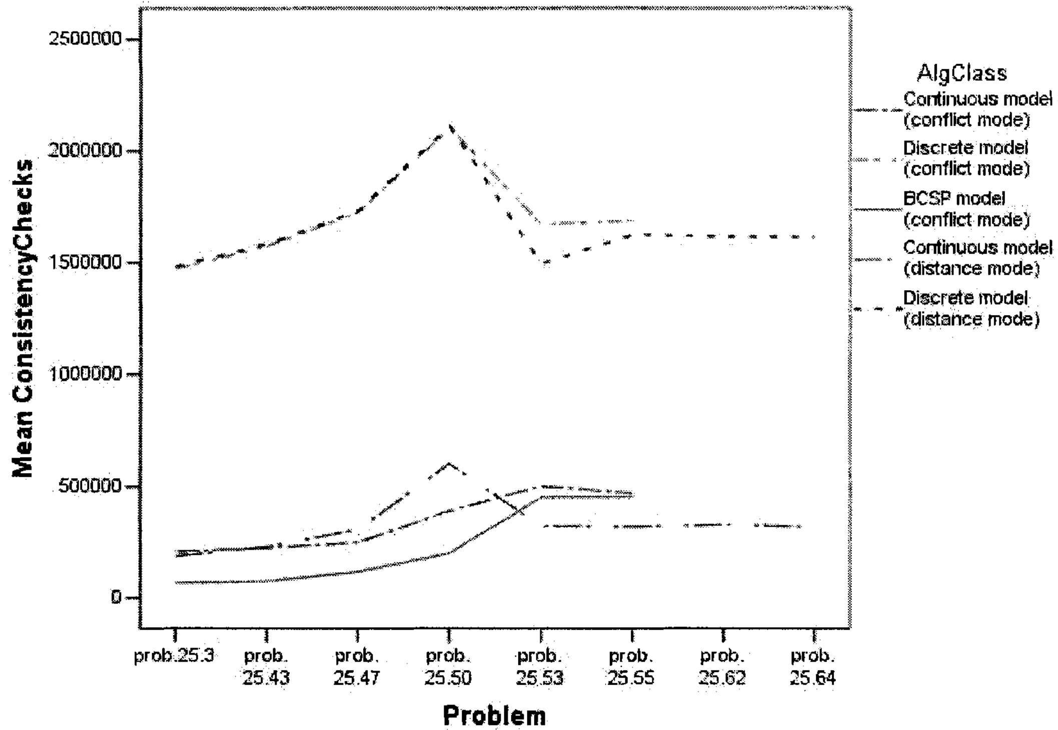


Figure 5.12: The mean number of consistency checks of PSO models from problem set 25.

1. The outcomes include PC configuration problem set 25 from the Comparison phase. See Figure C.8(a) and Figure C.8(b) for problem set 10 and problems 20.3~26.3, respectively.
2. The numbers of consistency checks of both the Continuous-conflict model and the BCSP model are relatively low and grow slowly. The BCSP model has lower number of consistency checks across all problems.
3. Both Discrete models have similar numbers of consistency checks between problems 25.3 and 25.50. The evaluation of the distance function is changed and the solutions are considered as 'acceptable' once its EV is smaller than 1 in problems 25.53, 55, 62 and 64. Therefore, the number of consistency checks of Discrete-distance then becomes lower than the one of Discrete-conflict.

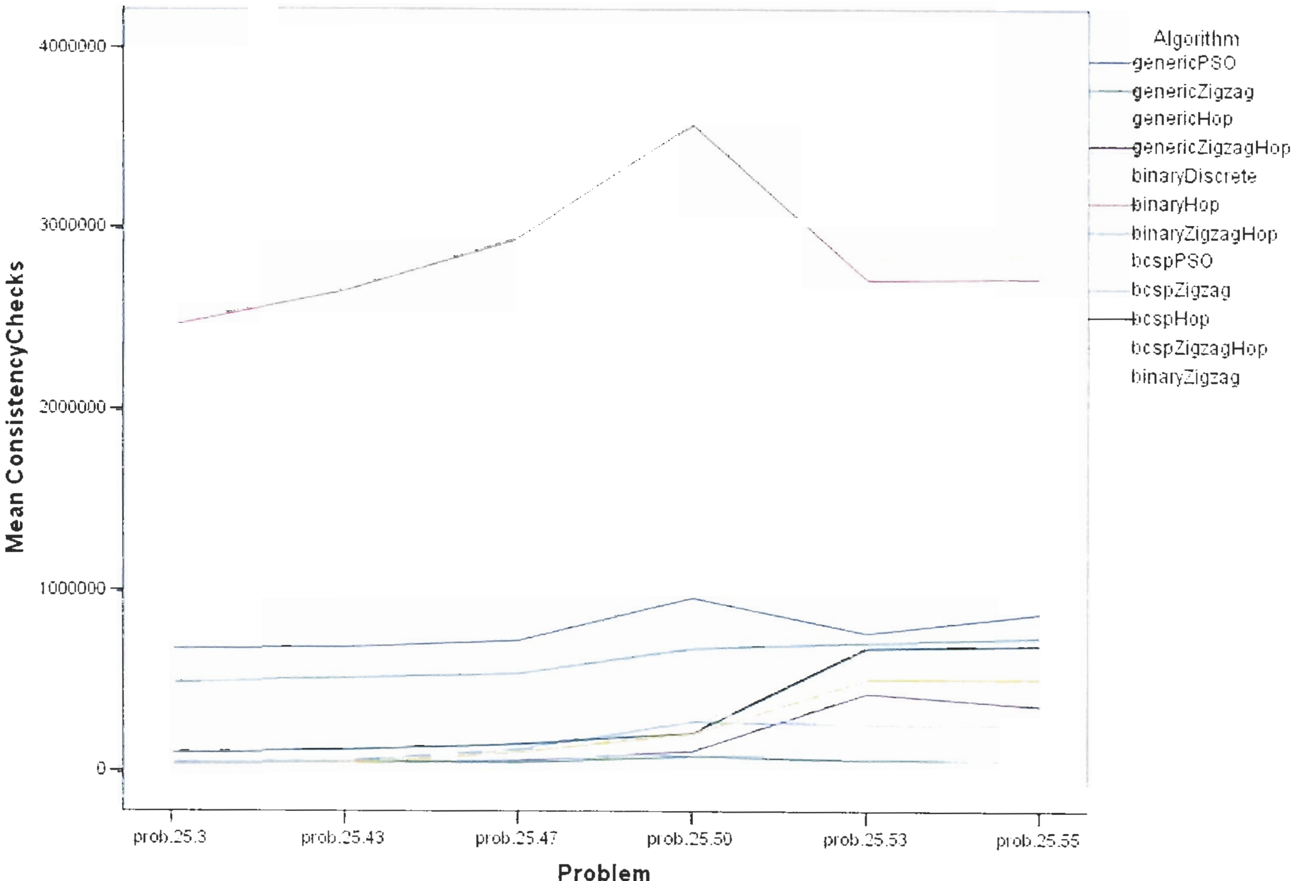


Figure 5.13: The mean number of consistency checks of PSO algorithms from problem set 25.

1. The outcomes present the individual PC configuration problems in problem set 25 from the Comparison phase. These algorithms use the conflict count function. See Figure C.9 and Figure C.10 for problem set 10 and problems 20.3∼26.3.
2. The algorithms can be divided into two groups. *binaryDiscrete* and *binaryHop* have the most number of consistency checks.
3. Since problem 25.50 has the most number of constraints, most of the algorithms more or less come to a peak at problem 25.50 except for *genericZigzag* and *genericZigzagHop*, which are not so obvious. Checking for the constraints on a particular variable at a time, most of the zigzag and zigzagHop type algorithms are pretty efficient and have much smoother lines compared to the hop type algorithms.

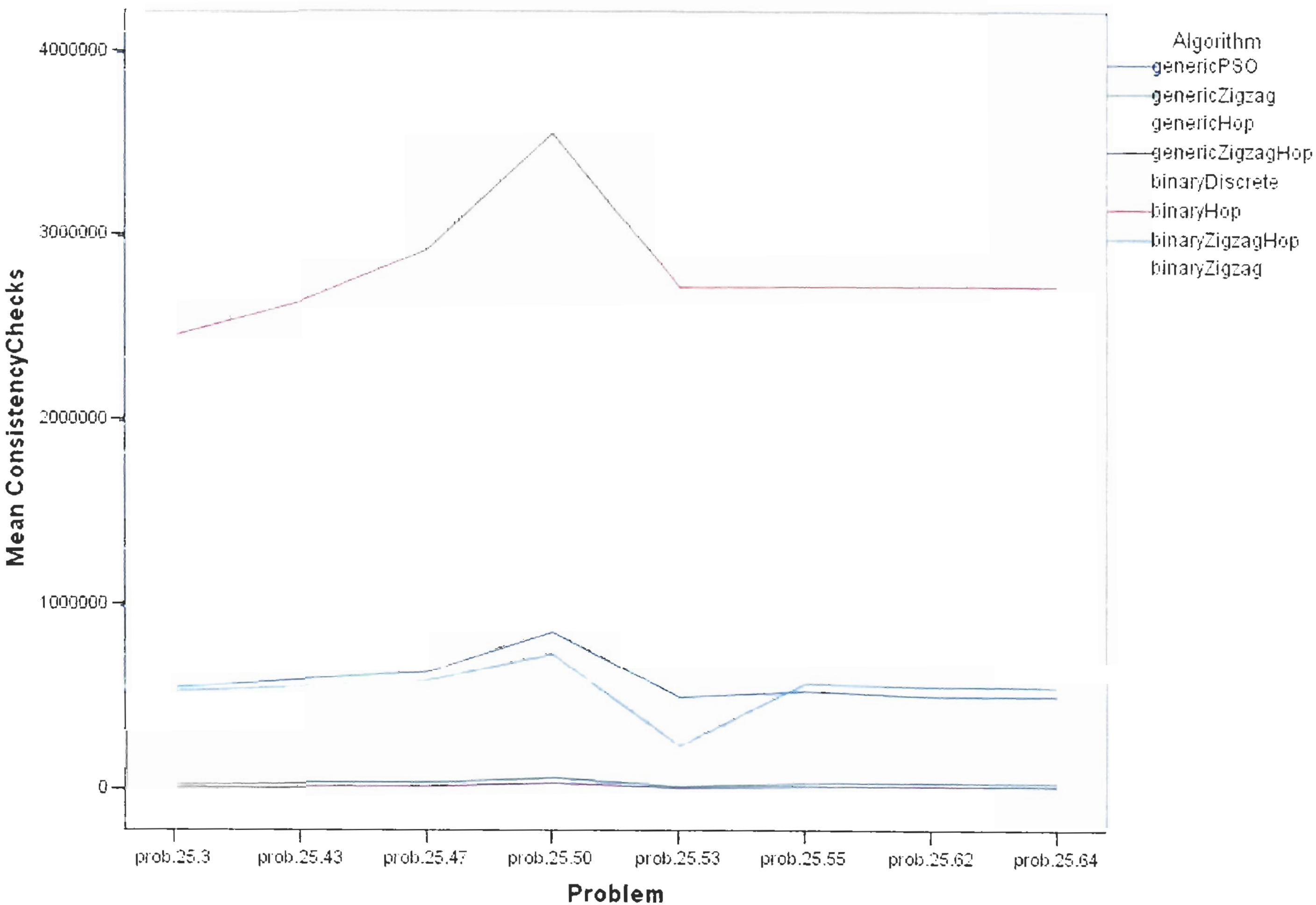
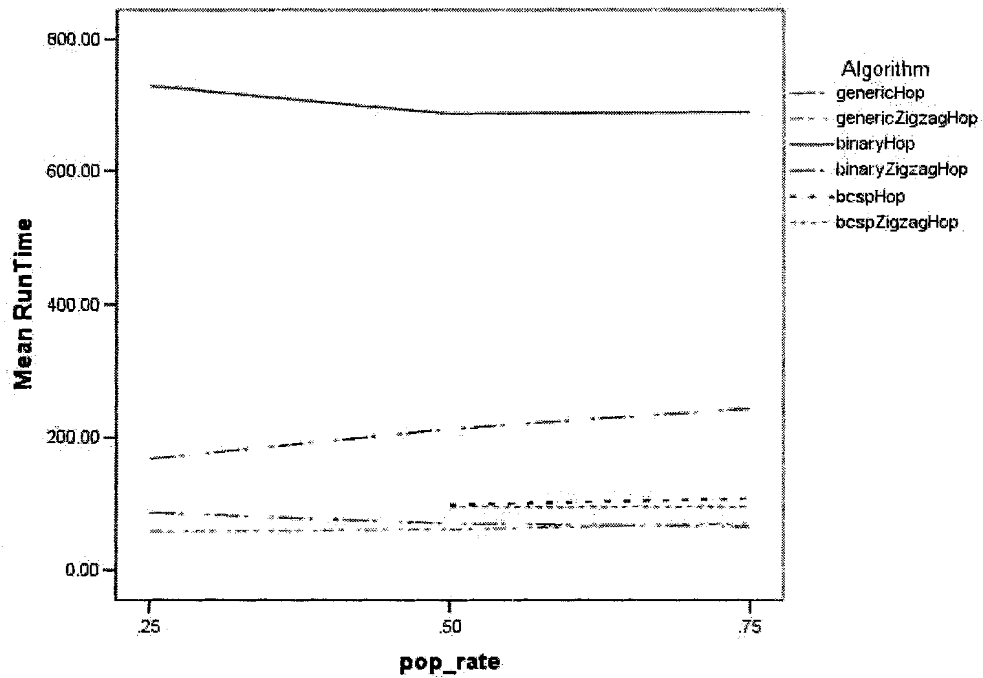
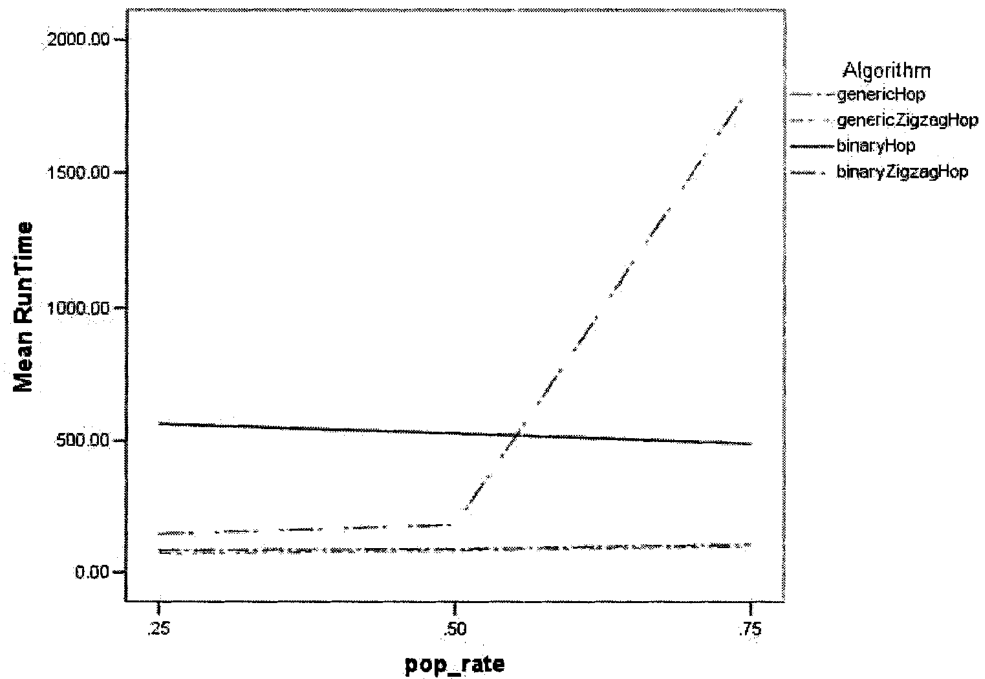


Figure 5.14: The mean number of consistency checks of PSO algorithms from the Comparison phase.

The outcomes present PC configuration problems in problem set 25. See Figure C.11 and Figure C.12 for problem set 10 and problems 20.3~26.3. respectively. These algorithms use the distance function.

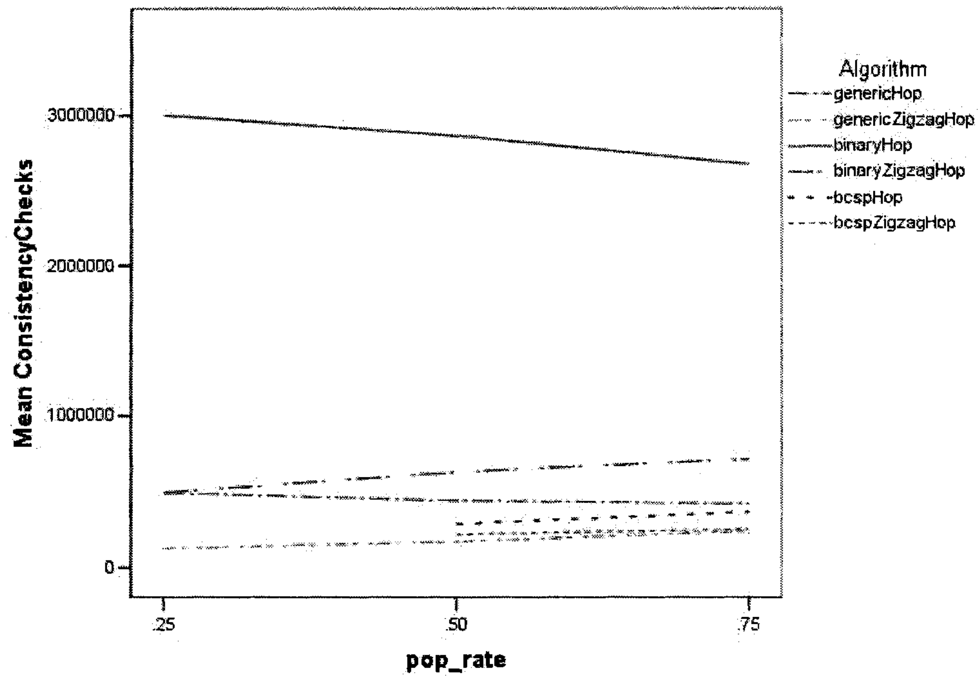


(a) Algorithms using conflict function

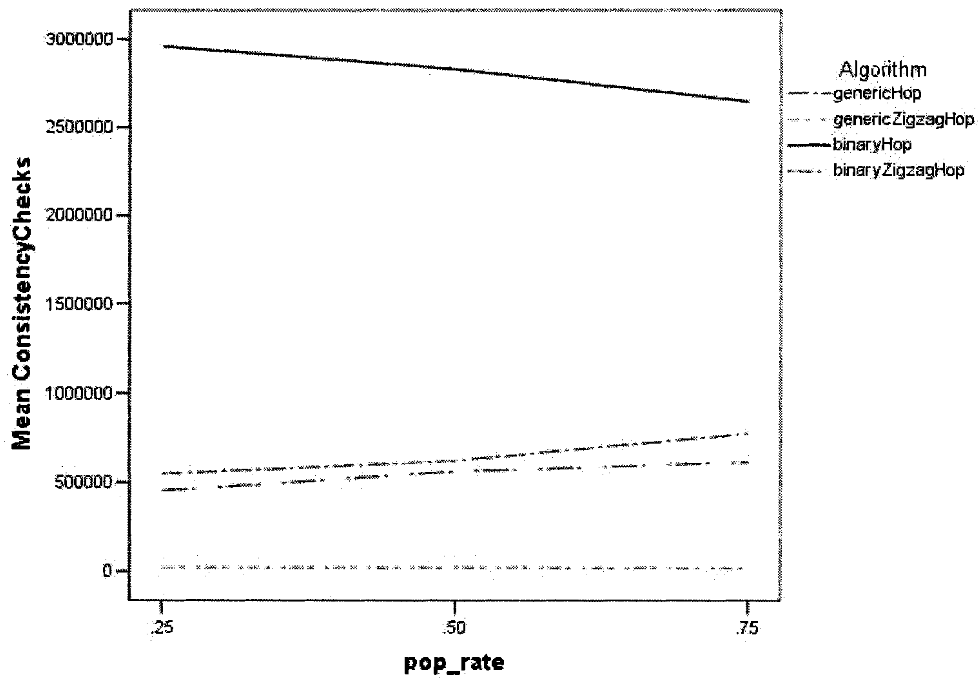


(b) Algorithms using distance function

Figure 5.15: The mean run time of PSO *pop_rate*: problem set 25 from the Comparison phase.



(a) Algorithms using conflict function



(b) Algorithms using distance function

Figure 5.16: Mean number of consistency checks of *pop_rate*: problem set 25 from Comparison phase.

5.5 Discussion and Answers

In this section, we will summarize the research results and discuss our research questions. Each of the following subsections corresponds to a research question stated in Section 5.1.

5.5.1 Can we extend Schoofs and Naudts' PSO to solve general n-ary integer CSPs effectively?

Schoofs and Naudts' original algorithm (*bcspPSO*) has difficulty in handling n-ary constraints ($n > 2$) and cannot solve n-ary CSPs effectively. The strategies we proposed, improve the performance of the *bcspPSO* algorithm. *bcspZigzagHop* extending the *bcspPSO* and combining the zigzag movement and no-hope and hop strategy, is one of the best performing algorithms in this research. It can reasonably solve n-ary CSPs effectively on non-hard problems.

5.5.1.1 Discussion – the BCSP PSOs

To solve binary CSPs (BCSP), Schoofs and Naudts extended the traditional PSO [53] with a set of operators and a conflict count objective function [90]. Based on their pseudocode, we implemented the algorithm as will be referred to as *bcspPSO*. Extending this algorithm, we developed three particle swarm algorithms: *bcspZigzag*, *bcspHop* and *bcspZigzagHop*.

The experimental results in Section 5.4.1.1, Section 5.4.1.2 and Section 5.4.2 show that Schoofs and Naudts' particle swarm (*bcspPSO*) cannot manage n-ary constraints in the PC configuration problem nor solve 4- or 5-queens test problems effectively. While reviewing the algorithm, we found that their vector-like operators and the conflict count function cause the ineffectiveness. In *bcspPSO*, a particle updates its position in each dimension[33] only from the choice of four: the global best position so far, its individual best position so far, its current position, or a random position. A random position may be chosen only when the corresponding variable (var_j) is in conflict and the particle's current position equals its individual best position so far. Even so, the probability of choosing between a random position and the global best position so far relies on the deflection operator,[34] which was set to either 0, $1/n$, or $2/n$ in the experiment. Since the chance for a random position to be

[33]A particle position is a complete CSP assignment and consists of n elements $\langle val_1, val_2, \ldots, val_n \rangle$. Each element is an assigned value to its corresponding CSP variable var_j where $j = 1, 2, \ldots, n$.

[34]See Section 3.5 and Section 4.2.1.3 as needed.

explored is rather low especially when n is big, the swarm has a hard time moving to other area to find a better solution.

As for the choice among the three other positions, it depends on whether the corresponding variable (var_j) is in conflict or not. If var_j is in conflict, the two best positions so far can be chosen; otherwise, the current position remains. Since the conflict count objective function cannot usually give accurate information on which variables are really in conflict in an n-ary constraint, all variables in the violated constraint are marked as in-conflict. If more variables are marked as in-conflict than there actually are, the swarm tends to converge to the global best position faster owing to the operation of the vector-like operators. This is obviously problematic to the algorithm while dealing with n-ary constraints because the information provided by the objective function is not as informative as while dealing with binary constraints. This quick convergence can be also observed from the fast run time of the algorithm although *bcspPSO* is very ineffective. Nothing much can be done after converging at local optima except reinitializing the algorithm whenever a no-hope count arrives or quickly looping through the remaining iterations. Even to reinitialize the algorithm, it may quickly converge to local optima again. So, the algorithm is generally faster than other particle swarm algorithms although it is ineffective in solving a problem.

Algorithms *bcspZigzag*, *bcspHop* and *bcspZigzagHop* improve upon the *bcspPSO* algorithm. *bcspZigzag* recomputes the constraint violations every step and provides more accurate information about the current positions than *bcspPSO* does. Empirically, we have shown that *bcspZigzag* improves the success rate of the *bcspPSO* algorithm. *bcspZigzag* does not appear to be more efficient overall than *bcspPSO* on run time. Running for the same time limit, *bcspPSO* does 20000 iterations, whereas *bcspZigzag* can do 50000. Algorithm *bcspHop* improves the *bcspPSO* algorithm even more, but the speed is a tradeoff as problems become harder. Unlike *bcspPSO* cycling between convergence and reinitialization, *bcspHop* spends time searching and improving solutions. As a CSP solver, *bcspZigzagHop* is not yet ready for solving hard problems. However, combining the merits of the zigzag movement and no-hope and hop strategy,[35] *bcspZigzagHop* improves the *bcspPSO* algorithm most. It is the best performing algorithm in this research, the most effective particle swarm algorithm in the `all_diff` experiment and has potential for further research.

[35] Refer to Section 4.2.2.1 and Section 4.2.2.3 as needed for the detail description.

5.5.2 How can we modify the traditional PSOs to solve n-ary integer CSPs? How do the algorithms extending the traditional PSOs compare with Schoofs and Naudts' PSO?

This research question includes two sub-questions and the answers to each of them involve two PSO models: the continuous PSO [53] and the discrete PSO [54]. We will discuss these two models in Section 5.5.2.1 and Section 5.5.2.2 respectively. The answers to the two questions are as follows:

1. Unlike Schoofs and Naudts' PSO (the BCSP PSO), the continuous PSO and the discrete PSO were not originally designed to solve CSPs. So, we have to first modify them so that the swarms understand and work with integer CSPs. In answering the question in Section 5.5.1, we have discussed some potential problems with Schoofs and Naudts' vector-like operators with n-ary constraints. Mathematically, the arithmetic computation of the original PSOs is simple and straightforward. Thus, we decided to keep the original formulae for updating particles' velocity and position. In order to restrain the particles from searching out of CSP domains, we relocate them to a closest *legal spot* as soon as they fly out of domain.[36]

2. The *genericZigzagHop* algorithm is the most effective algorithm among the continuous particle swarm algorithms in this research, and *binaryZigzagHop* is the one among the discrete algorithms. The *genericZigzagHop*-conflict algorithm[37] performs competitively to *bcspZigzagHop* on most of the test problems, but not so good as *bcspZigzagHop* on large problems. The *binaryZigzagHop* algorithm can be as effective as *bcspZigzagHop* as well, but the capability of *binaryZigzagHop* is more restricted by the domains of the constraint satisfaction problems. The *binaryZigzagHop* algorithm has difficulty with problems in which the domains are not consecutive or not consistent.[38]

---

[36] Refer to Section 4.2.1 as needed.

[37] Short for the *genericZigzagHop* using the conflict count objective function

[38] See Definition 2.1.1 for the definitions of a consecutive domain and consistent domains.

5.5.2.1 Discussion – the continuous PSOs

With the modification we have done, the continuous PSO can search discrete integer domains, and *genericPSO* is the basic implementation of the continuous particle swarm. Extending *genericPSO*, we developed a number of algorithms such as *genericZigzag*, *genericHop* and *genericZigzagHop*. Besides these algorithms, we also implemented other algorithms to experiment with several interesting strategies. For instance, the *genericHybrid* algorithm combines partner exchange, local depth-first search, and no-hope and hop strategies with the ability to spawn more particles.

The experimental results show that *genericPSO* is more effective than *bcspPSO* because *genericPSO* can explore more freely from its arithmetically computed particle positions and velocities rather than *bcspPSO*'s vector-like operation. The *genericPSO* algorithm tends to work better with consecutive CSP domains because of its continuous nature as indicated by its success rate of solving the PC configuration problems and the n-queens problems. However, *genericPSO* gets stuck at local optima quickly and is unable to escape from local optima. The zigzag movement alone improves the speed of the algorithm, but it does not improve the effectiveness very much.

Using the proposed strategies, the continuous particle swarm algorithms are able to solve the test problems with both consecutive and non-consecutive domains. For example, once the swarm has confined to a local optimum, *genericHop* repairs constraint violation to continue improving the solutions. *genericHybrid* also improves the performance of *genericPSO* but not as much as *genericHop* improves, which we have discussed in Section 5.4.1.1.

Implemented with the same strategies, the *genericZigzagHop*-conflict algorithm is the only swarm that can compete with *bcspZigzagHop*. The *genericZigzagHop*'s efficiency tends to be more stable across different problems, but *bcspZigzagHop* scales better on the sizes of the test problems. For example, *bcspZigzagHop* outperforms *genericZigzagHop*-conflict in solving the largest problems in the Comparison phase and the `all_diff` phase. *bcspZigzagHop*'s ability to restart the swarm besides the no-hope and hop strategy, may have contributed to its performance in solving these large problems. When we tried to reduce *genericZigzagHop*'s duration of no-hope cycle from $nohope = 2500$ to 1000 on problem 26.3, the experiment shows the potential to improve the algorithm by reducing its *nohope* count.

5.5.2.2 Discussion – the discrete PSOs

According to the authors of [54] and [55], two binary encodings can be used to encode integers and they suggest that Gray encoding works better than Binary encoding. We implemented the discrete particle swarms with both encodings[39] but we find that Gray encoding is not necessarily better than Binary encoding in the CSP context because CSP domains are not always consecutive over a range and they can be sparse as discussed in Section 4.2.1.2.

The discrete particle swarms perform well on the PC configuration problems in Formulation I and the n-queens problems; but, not being able to solve the test problems in Formulation II is a major drawback. Compared with the algorithms from the other PSO models, the effectiveness of the discrete algorithms greatly depends on the distribution of CSP domains. Since a discrete particle computes velocities and uses the velocities as probability thresholds to change its position bit string, the distribution of the CSP domain values and the consistency across all domains are important as discussed in Section 4.2.1.2. The domains of the test problems in Formulation I and the n-queens problems are rather consistent and consecutive, so the discrete algorithms are able to solve those problems more effectively.

As discussed in Section 3.3.3, each particle position in the Discrete model is a bit string of length xn if the domain value of a variable can be encoded in x bits. Regardless of the efficiency of the program implementation, the time complexity of the Discrete algorithms is x times of those of the continuous and BCSP algorithms for the same number of iterations. When the discrete particle swarms cannot solve a problem effectively and run up to the iteration limit, their total run time is high and inefficient.

The *binaryZigzagHop* algorithm improves *binaryDiscrete*'s success rate the most among the discrete particle swarm algorithms in this research. In solving PC configuration problem set 10 and the n-queens problems, it performs better than *genericZigzagHop*, but slightly less effective than *bcspZigzagHop*. Similar to the other discrete particle swarms, *binaryZigzagHop* could not solve any PC configuration test problems in Formulation II because the distribution of the CSP domains is not suitable for the algorithm.

[39]The algorithms with Binary encoding are *binaryDiscrete*, *binaryZigzag*, *binaryHop* and *binaryZigzagHop*; and the algorithms with Gray encoding are *grayDiscrete*, *grayZigzag* and *grayHop*

Chapter 6

Conclusion

In this research, we developed a number of particle swarm algorithms to solve general n-ary integer constraint satisfaction problems (CSPs) based on three existing particle swarm optimization (PSO) approaches. Among these three PSOs, one of them was developed to solve binary constraint satisfaction problems (BCSP) but not general n-ary CSPs. Although studies [80] show that it is possible to convert n-ary constraints to equivalent binary constraints, not all n-ary CSPs are suitable to be converted to equivalent binary ones in terms of the complexity of the problems. A CSP can become easier or harder to solve after the conversion [102], depending on the nature of the constraints. In addition, the process of converting an n-ary CSP to its equivalent binary CSP can be complicated and, not all the conversions can be done properly and produce semantically equivalent representation [47, 102]. Moreover, n-ary constraints provide a natural formulation for modelling real-world problems [86]. Thus, we did not limit our development for solving only binary CSPs. While developing the new particle swarm algorithms, we studied and modelled the relationship between the three PSOs and CSPs.

The two original PSOs [53, 54] were not originally designed for solving CSPs, so we first presented a way for the particle swarms to search through integer CSP domains. Extending the existing PSOs, we introduced algorithms appropriate for the CSP paradigm. For instance, some algorithms move the particles one dimension at a time in a zigzag style, which we have not seen before in the PSO research. With such movement, the particles can quickly step through the search space and evaluate more CSP assignments that differ only in some variables. Extending Schoofs and Naudts' no-hope and rehope mechanism [90] and CSP repair-based methods, we added a no-hope and hop technique to fix constraint violations

when the regular swarm stops improving the search. Both the zigzag movement and the no-hope and hop strategy outperform the original particle swarm algorithms in most of the test problems.

To handle constraint satisfaction, we made use of the PSO optimization mechanism and used two objective functions: conflict count and distance estimation. The distance function works well for arithmetic constraints, but it is more expensive and less effective than the conflict count function for evaluating goodlists, badlists and price constraints. Thus, the conflict count function is better in the PC configuration problem and the distance function is better in the n-queens problems where no goodlist or badlist constraints are used. However, none of these two functions provide information good enough to efficiently resolve n-ary constraints for large n.

In addition to developing the new particle swarm algorithms, we also used two different formulations to model a PC configuration problem as our major test problems in the Python CSP framework [20]. The first formulation is based on Tam and Ma's web-based configuration research [104]. With this formulation, the problem is simpler but requires a larger amount of preprocessing effort to make the data consistent. In the second formulation, we utilize the arithmetic relations provided by the Python CSP framework [20] to describe the constraints. This formulation gives much greater flexibility in representing the problem and we can describe the problem in more detail. In addition, we implemented an 'OR' constraint to enhance the Python CSP framework. With this disjunctive constraint, we can make CSP representation more flexible and more expressive although harder to solve.

6.1 Summary of the Research Results

In Section 5.5, we have discussed and answered the research questions based on the experimental results. From the results and discussions, we find that *bcspZigzagHop* is the most promising algorithm among the algorithms in this research as it has an average of 64% success rate on a set of configuration test problems and 76.7% on the 10-queens problem. The *genericZigzagHop* algorithm performing competitively to *bcspZigzagHop* with an average of 67% success rate on the same set of configuration problems, has the potential for further improvement, although it is not as good as *bcspZigzagHop* on large problems.

6.2 Future Work

Our research in developing particle swarm algorithms to solve general n-ary CSPs suggests many opportunities for future research. More thorough experimentation and more sophisticated n-ary constraint handling are needed. Other ideas for future research include:

- More intelligent repair methods can be used in the no-hope and hop strategy. For example, we can apply the min-conflict heuristic [65] to select a domain value. We have shown that random 'hops' improve the performance of the particle swarm algorithms. Applying more effective repair strategies, we may further enhance the algorithms.

- A better no-hope detection mechanism or more sophisticated diversity control as in [115] may help. Some incomplete experiments suggest that the *nohope* count we used in the experiment may not necessarily provide the right information for the repair strategies to take place. For instance, when we tried to reduce the duration of no-hope cycle of a particle swarm algorithm from $nohope = 2500$ to 1000, the experiment showed the potential for improving the results.

- The particles in the continuous and the discrete particle swarms only rely on the calculated velocities to update their positions. The BCSP particle swarms however, take constraint violation information into account for each variable. If a variable is not in conflict, a BCSP particle will not change its assignment to the variable. This is one of the reasons that the BCSP particle swarm algorithms perform more effectively and efficiently than the algorithms of the other two PSOs. We may put the same strategy into the continuous and discrete particle swarm algorithms to prevent particles from wasting efforts to update variable assignments that are not in conflict.

- In most of the algorithms of this research, each swarm consists of same types of particles, which all possess the same capability for solving CSPs. For example, in a zigzagHop type particle swarm, all the particles feature zigzag movement and the no-hope and hop strategy. However, different types of particles may have different capabilities to contribute. It will be interesting to see what interactions and collaboration may occur among different types of particles. For example, we can try to put several zigzag particles and several particles who know how to 'hop' together in a system, instead of all particles featuring both strategies. As a related extension, it is also possible to

build a particle swarm system which is able to learn a given problem, find the most suitable particles, and adaptively adjust its settings to solve the problem.

- We mentioned that some experiments on several strategies were not completed owing to time limitation. Some of these strategies, such as the piggy bank strategy and diversity control described in Section 4.2.2.4 and 4.2.2.5, may be worth further research.

- Developing a particle swarm visualization system [91, 26] may help us understand how the swarm searches through the CSP search space. We started a simple 2-dimensional system, which only takes 2-variable problems. To develop a visualization system that can present an n-variable CSP, we need to resolve two major challenges. One is to represent an n-dimensional space using an x-y plane, and another is to visualize n-ary constraints.

- To be practical, the studies on the applicability of the algorithms should be done and several possible directions are:

  - In addition to a PC configuration problem, many real-world problems can be used such as scheduling problems, resource allocation, and so on. Also, taking an actual real-world sized problem can be useful too.

  - Applying the algorithms to the real-world problems, speed is an important issue. Our experiments were only limited to several sets of parameter settings. Research on tuning the parameter settings and improving the efficiency of the algorithms can be helpful.

  - Comparing particle swarm algorithms with other CSP algorithms is also an important subject.

- Since PSO is able to start with any initial solutions and return a potential best solution so far at any time, we may further extend our research results to dynamic environment.

Appendix A

Algorithms and Examples

A.1 CSP Examples in Python CSP Framework

```
# define variables:
v = var(1, 11) # i.e. a list of domain [1, 2, 3, ... 10]

# create a CSP:
csp = problem(v)

# add constraints
csp += even(v)
```

Figure A.1: The warm-up example of Section 2.2.1 in the Python CSP framework.

```
# define variables:
a, b, c = var(1, 31), var(1, 31), var(1, 31)

# create a CSP:
csp = problem(a, b, c)

# add constraints
csp += a**2 + b**2 == c**2
```

Figure A.2: The Pythagorean triple example of Section 2.2.2 in the Python CSP framework.

```
# define variables:
q0, q1, q2, q3 = var(0, 8), var(0, 8), var(0, 8), var(0, 8)
q4, q5, q6, q7 = var(0, 8), var(0, 8), var(0, 8), var(0, 8)

# create a CSP:
q = [q0, q1, q2, q3, q4, q5, q6, q7]
csp = problem(*q)

# add constraints
csp += all_diff(*q)
for i in xrange(8):
    for j in xrange(8):
        if i != j:
            csp += q[i] - q[j] != abs(j - i)
```

Figure A.3: 8-Queens problem of Section 2.2.3 in the Python CSP framework.

Notation '*q' is a feature of Python, for which one place a list of any number of variables as needed; for instance csp = problem(*q) is equivalent to csp = problem(q0, q1, q2, q3, q4, q5, q6, q7).

```
# define variables:
s, e, n, d = var(1, 10), var(0, 10), var(0, 10), var(0, 10)
m, o, r, y = var(1, 10), var(0, 10), var(0, 10), var(0, 10)

# create a CSP:
csp = problem(s, e, n, d, m, o, r, y)

# add constraints
csp += all_diff(s, e, n, d, m, o, r, y)
csp += 1000*s + 100*e + 10*n + d +1000*m + 100*o + 10*r + e \
       == 10000*m + 1000*o + 100*n + 10*e + y
```

Figure A.4: The send-more-money puzzle of Section 2.2.4 in the Python CSP framework.

```
# define variables:
# enumerate color red = 1, green = 2, blue = 3
# range(1,4) render a list [1,2,3]
R1, R2, R3 = var(range(1,4)), var(range(1,4)), var(range(1,4))
R4, R5 = var(range(1,4)), var(range(1,4))

# create a CSP:
csp = problem(R1, R2, R3, R4, R5)

# add constraints
csp += R1 != R2
csp += R1 != R4
csp += R1 != R5
csp += R2 != R3
csp += R2 != R4
csp += R3 != R4
csp += R3 != R5
csp += R4 != R5
```

Figure A.5: The sample graph colouring problem of Section 2.2.5 in the Python CSP framework.

A.2 Swarm Algorithms

```
PSO(problem, P, max_iter, c1, c2, F)
    comments: for minimization
 1  gbest ← initialized to some very big value
 2  t ← 0
 3  x_i[t] ← initialize particle's position
 4  v_i[t] ← initialize particle's velocity
 5  pbest_i ← initialized to some very big value

 6  while t < max_iter
 7       do for i ← 1 to length[P]
 8              do eval ← evaluate x_i[t] with some objective function F
                   comments: update pbest_i and gbest if it is appropriate
 9                 if eval < pbest_i
10                    then pbest_i ← eval
11                         xpbest_i ← x_i[t]
12                 if eval < gbest
13                    then gbest ← eval
14                         xgbest ← x_i[t]
15          t ← t + 1
16          for i ← 1 to length[P]
17              do r1, r2 ← random(), random()
                   comments: calculate velocity
18                 v_i[t] ← v_i[t − 1] + r1c1(xpbest_i − x_i[t]) + r2c2(xgbest − x_i[t])
                   comments: update position
19                 x_i[t] ← x_i[t − 1] + v_i[t]
20  return gbest, xgbest
```

Figure A.6: Pseudocode of the continuous PSO with global best information [55]

This pseudocode makes reference to page 296 in [55], but in more detail. Also, our goal is to find an minimum whereas the one in [55] is to find a maximum.

```
DISCRETEPSO(problem, P, dimension, max_iter, c1, c2, F)
    comments: for minimization
 1  gbest ← initialized to some very big value
 2  t ← 0
 3  x_i[t] ← initialize particle's position to some bitstring
 4  v_i[t] ← initialize particle's velocity
 5  pbest_i ← initialized to some very big value

 6  while t < max_iter
 7      do for i ← 1 to length[P]
 8              do eval ← evaluate x_i[t] with some objective function F
                    comments: update pbest_i and gbest if it is appropriate
 9                  if eval < pbest_i
10                     then pbest_i ← eval
11                          xpbest_i ← x_i[t]
12                  if eval < gbest
13                     then gbest ← eval
14                          xgbest ← x_i[t]

15         t ← t + 1
16         for i ← 1 to length[P]
17              do for d ← 1 to dimension
18                     do r1, r2 ← random(), random()
                          comments: calculate velocity
19                        v_id[t] ← v_id[t − 1] + r1 c1 (xpbest_id − x_id[t])
                                    +r2 c2 (xgbest_d − x_id[t])
                          comments: update position
20                        if random() < sigmoid(−v_ij(t)) ## reference to [54, 55]
21                           then x_id[t] ← 1
22                           else x_id[t] ← 0
23  return gbest, xgbest
```

Figure A.7: A pseudocode of a discrete version [54] of the PSO in Figure A.6

```
PSOforCSPs(problem, P, max_iter, φ1, φ2, deflection, noHope)
 1  randomly initialize the particles
 2  initialize gbest, all lbest's and all pbest's, and xgbest, all xlbest's and all xpbest
 3  t ← 1
 4  while t < maximum number of iterations:
 5        do for i ← 1 to population:
 6              do for j ← 1 to n:
 7                    do nbConf ← conflict counts of x_ij[t − 1] of particle P_i
 8                       if nbConf > φ1:
 9                          then v' ← xpbest_ij ⊖ x_ij[t − 1]
10                          else  v' ← x_ij[t − 1] ⊖ x_ij[t − 1]
11                       if nbConf > φ2:
12                          then if random() < deflection:
                                   #comments: it was 'if deflection' in [90]#
13                                 then v'' ← Rand(j) ⊖ x_ij[t − 1]
14                                 else  v'' ← xgbest_j ⊖ x_ij[t − 1]
15                          else  v'' ← x_ij[t − 1] ⊖ x_ij[t − 1]
16                       x_ij[t] ← x_ij[t − 1] ⊕ (v' ∘ v'')
17              fitness_i ← conflict counts in particle P_i
18              if fitness_i < pbest_i:
19                 then xpbest_i, pbest_i ← x_i, fitness_i
20              if pbest_i does not change for noHope times:
21                 then randomly initialize x_i
22        gbest, xgbest ← update from pbest, xpbest
23        lbest, xlbest ← update from pbest, xpbest
24        t ← t + 1
25  return gbest, xgbest
```

Figure A.8: Schoofs and Naudts' PSO for solving binary CSPs [90], named as *bcspPSO* in this research.

It serves as the foundation of all algorithms derived from BCSP model in this research.

- *gbest* is the global best fitness, *lbest* keeps the local best fitness values of all swarm neighbourhoods, and *pbest* keeps the individual best fitness values of all particle.
- *xgbest* is the global best position, *xlbest* keeps the local best positions of all swarm neighbourhoods, and *xpbest* keeps the individual best positions of all particle.
- *population* is the number of particles of the swarm.
- φ_1 and φ_2 are some coefficients to determine that the velocity update relies more on global best experience or on individual best experience.
- *Rand(j)* randomly returns a value from the domain D_j.
- *noHope* is an upper bound defined for determining when there is no hope for the swarm to improve the solution and a no-hope mechanism should come in.

A.3 Particle Swarm Algorithms for Solving CSPs

A.3.1 Local depth-first search: genericDFS

In Section 4.2.2.7, we mentioned that we need to impose some control upon *genericDFS* when we combine a local depth-first search with *genericPSO*.

Firstly, we divide all n CSP variables into several groups to keep the local DFS manageable. If there are enough[1] variables to be distributed among p particles, the n variables can be simply divided into p groups. Otherwise, we may permute the variables to produce p groups of variables such that no two groups are the same.[2] Each particle is assigned to a group of variables to perform a local DFS. The former case is straightforward, but the latter may require some explanation. For example, suppose we have a swarm of 9 particles to perform DFS on some best solution so far and the problem consists of 5 variables. There are obviously not enough variables to be distributed among the 9 particles, and so we permute the variables and group them as shown in Figure A.9. While the swarm is executing local DFS, each particle performs DFS only on the variables allocated to it (i.e. the **DFS variables** assigned to the particle) but not on the remaining **non-DFS** variables.[3] For instance, while a particle is performing DFS on the DFS variables $\{var_4$, var_5 and $var_1\}$ in Figure A.10, the assignments of non-DFS variables remain the same. Since the size of a DFS variable group is limited e.g. 3 or 4 variables, the execution time can be manageable. We may use a graph colouring problem shown in Figure 2.5 to illustrate how the system works. Five variables var_1, var_2, ..., and var_5 correspond to the five regions $R1$, $R2$, ..., and $R5$. For particles P_1, P_2, ..., P_9 to perform DFS, these 5 variables are arranged into 9 different variable groups of size 3 as shown in Figure A.12.

Besides randomly grouping the variables, we can also arrange the *related* variables together to improve the effectiveness of the complete search. That is, if there exist a number of constraints among some variables, we should first consider putting these variables in a group by applying the variable ordering techniques discussed in Section 2.3.3.3.

---

[1] At least there are 2 to 3 variables per group but not too many so that the DFS can be meaningful and manageable. We set the DFS size to 3 variables in our experiments.

[2] In effect, we can minimize the chance of evaluating the same potential solution. Nevertheless, if we cannot avoid duplicates because of too many particles or too few variables, some overlaps may still be useful because the non-DFS variables for each particle may not necessarily have assigned to the same values as shown in Figure A.11 and ended up different solution states.

[3] Each particle still evaluates the entire potential solution at each state to see if it finds some solution better in its local DFS.

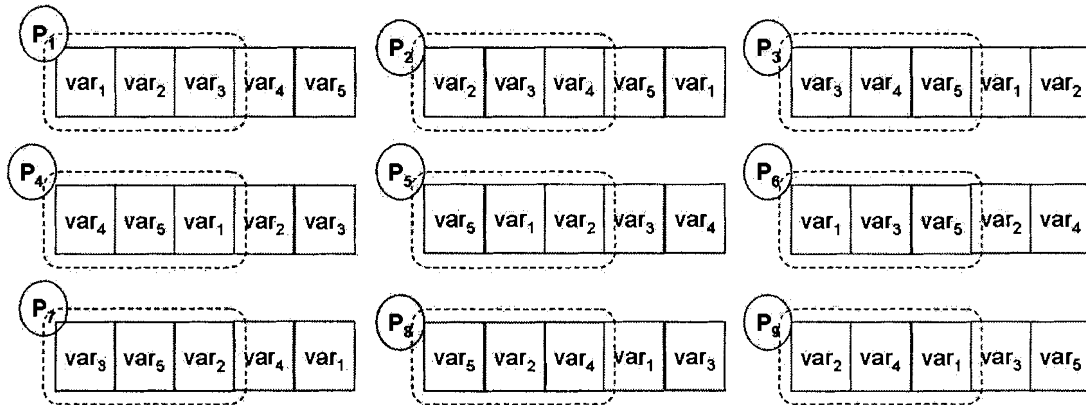


Figure A.9: Distribute 5 variables to 9 particles for performing DFS.

5 variables cannot be distributed into 9 groups, so we may permute these 5 variables to produce at least 9 different permutations. Since we want each particle to take on 3 (DFS) variables, the first three variables of each permutation must be different so that no two particles are responsible for the same variables.

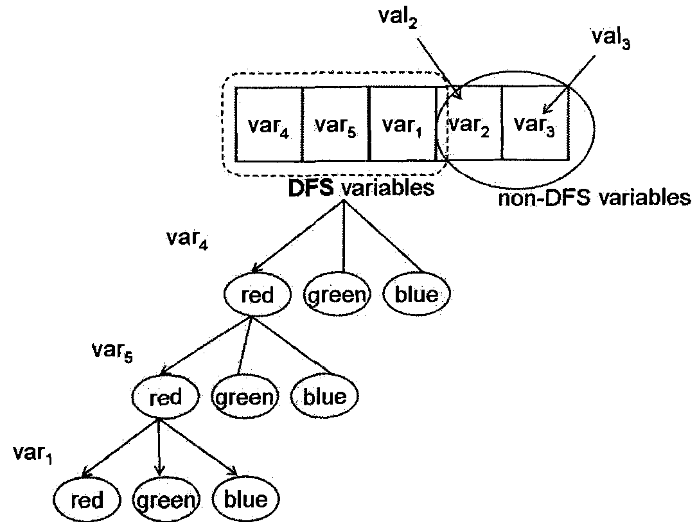


Figure A.10: A particle performs depth-first search on variable set $\{var_4, var_5, var_1\}$

While a particle is performing depth-first search on its DFS variables $\{var_4, var_5, var_1\}$, the values (val_2 and val_3) of non-DFS variables remain. Whenever a state is generated, the particle evaluates the entire potential solution to see if it finds some solution better. If it does find a better solution, it updates the best solution so far *gbest* as a regular PSO. Then, it will continue DFS until the local DFS is completed.

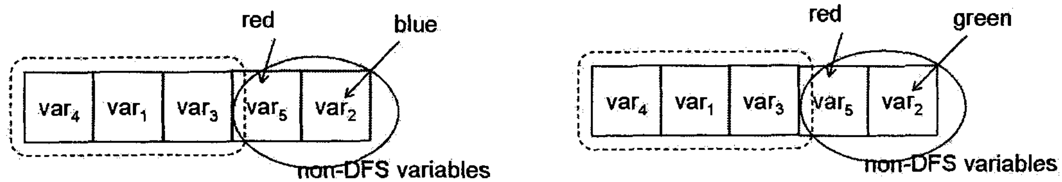


Figure A.11: Different non-DFS variable values generate different assignments in a graph colouring.

If we cannot avoid duplicates because of too many particles or too few variables, some overlaps may still be useful because the non-DFS variables for each particle may not necessarily have assigned to the same values and ended up with different solution states.

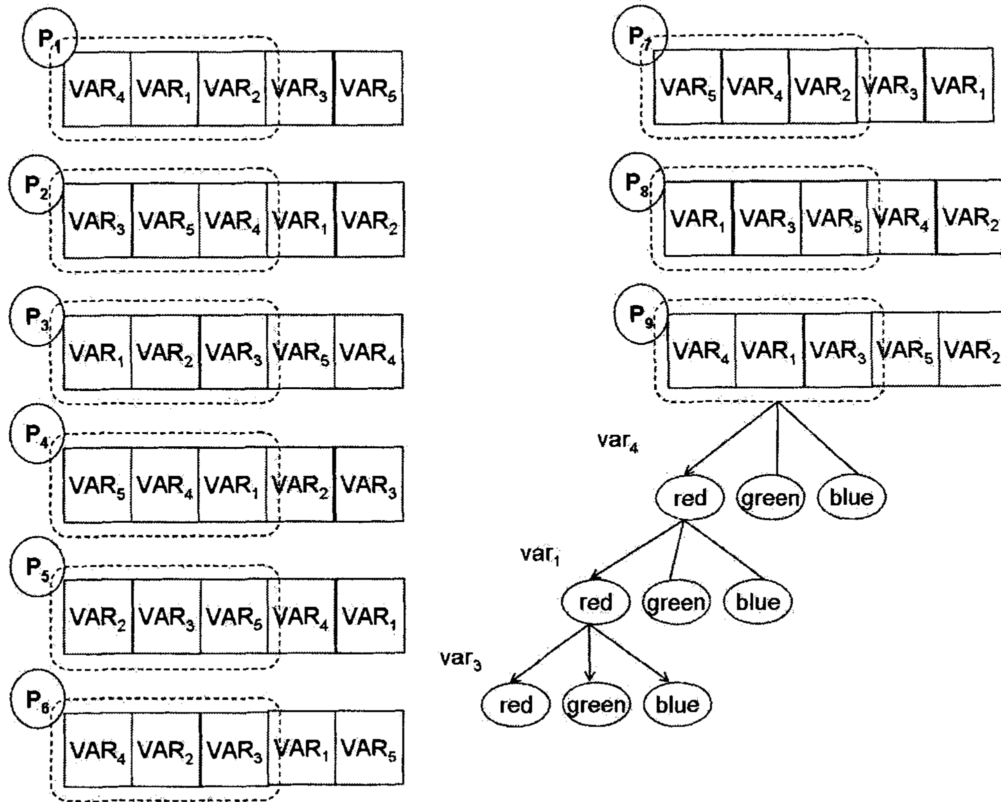


Figure A.12: Particles perform depth-first search in the graph colouring problem.

Five variables var_1, var_2, ..., var_5 correspond to the five regions $R1$, $R2$, ..., $R5$ in Figure 2.5. For particles P_1, P_2, ..., P_9 to perform DFS, these 5 variables are arranged into 9 different variable groups of size 3. Typically, these "3-DFS variable" groups are all different. Each particle is assigned to 3 DFS variables and 2 non-DFS variables, and it is responsible for running DFS on the DFS variables as shown for particle P_9 here.

Appendix B

PC Configuration Test Problems

B.1 Formulation I

B.1.1 The variables and the domains

A collection of the enumerated values is the domain of that component variable.

Table B.1: Sample CPUs for var_{cpu}

Component specification	Enumeration
AMD ATHLON 64 3000+ 2.0GHz S754 800fsb	0
AMD Mobile ATHLON XP-M 2500+ 1.86GHz SOCKETA 266fsb	1
INTEL PENTIUM 4 3.0GHz S478 800fsb	2
AMD ATHLON 64 3200+ 2.2GHz S754 800fsb	3
AMD ATHLON 64 3500+ 2.2GHz S939 2000fsb	4
AMD SEMPRON 2500+ 1.75GHz SOCKETA 333fsb	5
INTEL PENTIUM 4 2.8GHz S478 800fsb	6
INTEL PENTIUM 4 3.0GHz S478 800fsb	7
AMD ATHLON 64 2800+ 1.8GHz S754 1600fsb	8
INTEL PENTIUM 4 3.2GHz S478 800fsb	9

Table B.2: Sample RAMs for var_{ram}

Component specification	Enumeration
SAMSUNG 512MB 184pin PC3200	0
CORSAIR 1024MB 184pin PC3200 dual	1
OCZ 512MB 184pin PC3200	2
CORSAIR 1024MB 184pin PC3200 dual	3
SAMSUNG 256MB 184pin PC3200	4
KINGSTON 1024MB 184pin PC3200 dual	5
OCZ 512MB 184pin PC3200 dual	6
OCZ 1024MB 184pin PC3200 dual	7
INFINEON 512MB 184pin PC3200	8
KINGSTON 512MB 184pin PC3700 dual	9

Table B.3: Sample motherboards for var_{mb}

Component specification	Enum
SOLTEK SOCKETA dual RAM 184pin (400,333,266)fsb, onboard, IDE, AGP, PCI, USB	0
ASUS SOCKETA dual RAM 184pin (400)fsb, onboard, AGP, PCI, USB, WLAN, Firewire	1
ASUS S754 184pin (800)fsb, onboard, IDE, AGP, PCI, USB, WLAN, Firewire	2
ASUS S478 dual RAM 184pin (800,533,400)fsb, onboard, IDE, AGP, PCI, USB, WLAN	3
SOLTEK S754 184pin (800)fsb, onboard, IDE, AGP, PCI, USB	4
MSI S754 184pin ?fsb, onboard, IDE, AGP, PCI, USB, Firewire	5
ASUS S478 dual RAM 184pin (800,533,400)fsb, onboard, IDE, AGP, PCI, USB, WLAN, Firewire	6
ASUS S754 184pin ?fsb, onboard, AGP, PCI, USB, Firewire	7
ASROCK SOCKETA 184pin (333,266,200)fsb, onboard, IDE, AGP, PCI, USB	8
ASUS SOCKETA 184pin (400,333,266,200)fsb, onboard, IDE, AGP, PCI, USB	9

Table B.4: Sample VGAs for var_{vga}

Component specification	Enumeration
ATI RADEON X800 256MB (VGA,DVI,TV)out	0
MSI RADEON 9800 PRO 128MB (VGA,DVI,TV)out	1
LEADTEK GEFORCE 6800 128MB (VGA,DVI,TV)out	2
ATI RADEON 9800 PRO 256MB (VGA,DVI,TV)out	3
LEADTEK GEFORCE 6800 256MB (VGA,DVI,TV)out	4
ATI RADEON 9800 PRO 128MB (VGA,DVI,TV)out	5
SAPPHIRE RADEON 9550 128MB (VGA,DVI,TV)out	6
MSI GEFORCE 6800 Ultra 256MB (DVI,TV)out	7
SAPPHIRE RADEON 9800 PRO 128MB (VGA,DVI,TV)	8
BFG GEFORCE 6800 256MB (VGA,DVI,SV)out	9

Table B.5: Sample sound cards for var_{snd}

Component specification	Enumeration
CREATIVE	0
CREATIVE	1
CREATIVE	2
CREATIVE	3
CHAINTECH	4
M-AUDIO	5
CREATIVE	6
AOPEN	7
M-AUDIO	8
CREATIVE	9

Table B.6: Sample NICs for var_{nic}

Component specification	Enumeration
D-LINK (10,100)mbps PCI	0
SMC (10,100)mbps PCI	1
MICRONET (10,100,1000)mbps PCI	2
INTEL (10,100,1000)mbps PCI	3
D-LINK (10,100)mbps PCMCIA	4
INTEL (1000)mbps PCI	5
3COM (10,100)mbps PCI	6
SURECOM (10,100)mbps PCI	7
LINKSYS (10,100)mbps USB	8
LINKSYS (10,100)mbps PCI	9

Table B.7: Sample floppy drives for var_{fdd}

Component specification	Enumeration
MITSUMI BLACK	0
MITSUMI IVORY	1
Generic BLACK	2
Generic IVORY	3
SONY External BLACK	4
SONY External BLACK	5
ASUS IVORY	6

Table B.8: Sample hard drives for var_{hdd}

Component specification	Enumeration
WD 200MB 7200rpm 8mb IDE	0
SEAGATE 200MB 7200rpm 8mb IDE	1
SEAGATE 200MB 7200rpm 8mb SATA	2
SEAGATE 80MB 7200rpm 2mb IDE	3
SEAGATE 120MB 7200rpm 8mb SATA,IDE	4
SEAGATE 120MB 7200rpm 8mb IDE	5
MAXTOR 80MB 7200rpm 2mb IDE	6
WD 74MB 10000rpm 8mb SATA	7
WD 36MB 10000rpm 8mb SATA	8
WD 80MB 7200rpm 8mb IDE	9

Table B.9: Sample CD-ROMs for var_{cd}

Component specification	Enumeration
LG IVORY r EIDE	0
LITEON BLACK r EIDE	1
LG BLACK r EIDE	2
BENQ BEIGE r EIDE	3
ASUS BLACK r EIDE	4
SONY IVORY r EIDE	5
ASUS GREY r EIDE	6
LITEON IVORY r EIDE	7
MSI IVORY r EIDE	8
AOPEN WHITE r EIDE	9
LG WHITE r+w EIDE	10
LITEON IVORY r+w EIDE	11
LG BLACK r+w EIDE	12
TOSHIBA BEIGE r+w EIDE	13
LG BLACK r+w+dvd EIDE	14
BENQ BEIGE r+w EIDE	15
LG IVORY r+w+dvd EIDE	16
LITEON BLACK r+w EIDE	17
SONY WHITE r+w EIDE	18
LITEON BEIGE r+w+dvd EIDE	19

Table B.10: Sample power supplies for var_{power}

Component specification	Enumeration
THERMALTAKE 420w 12V ATX PS/2	0
OCZ 520w 12V ATX PS/2	1
ANTEC 480w 12V ATX PS/2	2
ENERMAX 350w 12V ATX PS/2	3
SPARKLE 300w 12V ATX PS/2	4
generic 350w 12V ATX PS/2	5
ANTEC 430w 12V ATX PS/2	6
ANTEC 480w 12V ATX PS/2	7
ULTRA 500w 12V ATX PS/2	8
ANTEC 550w 12V ATX PS/2	9

Table B.11: Sample casings for var_{tower}

Component specification	Enumeration
ANTEC BLACK ATX 380w, 5.25,3.5,USB,Firewire	0
ANTEC BLACK ATX 350w, 5.25,3.5,USB,PS/2	1
TSUNAMI IVORY ATX 400w, 5.25,3.5,USB	2
RAIDMAX BLACK ATX 420w, 5.25,3.5	3
ANTEC SILVER ATX no, 5.25,3.5,USB	4
ANTEC silver+black MicroATX 300w, 5.25,3.5,USB,Firewire	5
ASPIRE black,silver,blue,green,yellow ATX 350w, 5.25,3.5,USB	6
NGEAR BLACK+SILVER ATX 350w, 5.25,3.5	7
ANTEC BRONZE ATX 300w, 5.25,3.5,USB,PS/2	8
NGEAR BEIGE ATX 350w, 5.25,3.5,USB	9

Table B.12: Sample mice for var_{mouse}

Component specification	Enumeration
LOGITECH USB RED optical	0
LOGITECH USB,PS/2 BLUE,RED,SILVER wireless	1
LOGITECH optical wireless	2
LOGITECH USB,PS/2 BLUE+SILVER optical	3
LOGITECH USB,PS/2 SILVER+BLACK optical	4
LOGITECH USB,PS/2 SILVER+BLACK optical	5
LOGITECH USB,PS/2 WHITE optical	6
MICROSOFT USB SILVER+BLACK	7
QTRONIX USB,PS/2 SILVER+BLACK optical	8

Table B.13: Sample monitors for var_{scr}

Component specification	Enumeration
SAMSUNG 17 crt 1280x1024 IVORY D-Sub	0
SAMSUNG 17 crt 1280x1024 SILVER+BLACK D-Sub	1
LG 17 crt 1280x1024 BLACK D-Sub	2
LG 17 crt 1280x1024 WHITE D-Sub	3
LG 17 crt 1280x1024 IVORY D-Sub	4
VIEWSONIC 17 crt 1280x1024 SILVER+BLACK D-Sub	5
VIEWSONIC 17 crt 1280x1024 BLACK D-Sub	6
NEC 17 crt 1280x1024 WHITE D-Sub	7
VIEWSONIC 17 crt 1280x1024 WHITE D-Sub	8
VIEWSONIC 17 crt 1920x1440 IVORY D-Sub	9
SAMSUNG 19 crt 1600x1200 WHITE D-Sub	10
MITSUBISHI 19 crt 1920x1440 BLACK D-Sub	11
VIEWSONIC 19 crt 2048x1536 BLACK D-Sub	12
VIEWSONIC 19 crt 1600x1200 BLACK D-Sub	13
LG 19 crt 2048x1536 IVORY D-Sub	14
SAMSUNG 19 crt 1600x1200 BLACK+SILVER D-Sub	15
AOC 19 crt 1600x1200 WHITE D-Sub	16
SAMSUNG 19 crt 1920x1440 SILVER D-Sub	17
VIEWSONIC 19 crt 2048x1536 WHITE D-Sub	18
VIEWSONIC 19 crt 1600x1200 IVORY D-Sub	19
BENQ 19 lcd 1280x1024 SILVER+BLACK D-Sub,DVI	20
BENQ 17 lcd 1280x1024 SILVER+BLACK D-Sub,DVI	21
BENQ 17 lcd 1280x1024 SILVER+BLACK D-Sub,DVI	22
BENQ 15 lcd 1024x768 BLACK+SILVER D-Sub,DVI	23
BENQ 17 lcd 1280x1024 BLACK+SILVER D-Sub,DVI	24
VIEWSONIC 15 lcd 1024x768 SILVER+BLACK D-Sub,DVI	25
SAMSUNG 19 lcd 1280x1024 BLACK D-Sub,DVI	26
SAMSUNG 17 lcd 1280x1024 SILVER D-Sub,DVI	27
SAMSUNG 17 lcd 1280x1024 BLACK D-Sub,DVI	28
LG 17 lcd 1280x1024 SILVER D-Sub,DVI	29

Table B.14: Sample printers for var_{prt}

Component specification	Enumeration
CANON 4800x1200 Parallel,USB,DPP color	0
CANON 4800x1200 USB,DPP color	1
CANON 4800x1200 USB color	2
CANON 4800x1200 Parallel,USB,DPP color	3
LEXMARK 4800x1200 USB color	4
EPSON 5760x1440 USB color	5
EPSON 5760x1440 Parallel,USB color	6
CANON 4800x1200 USB color	7
HP 4800x1200 USB color	8
HP 4800x1200 color	9
BROTHER 1200x600 Parallel,USB laser b/w	10
SAMSUNG 600x600 Parallel,USB laser b/w	11
SAMSUNG 1200 Parallel,USB laser color	12
HP laser color	13
SAMSUNG 1200x600 Parallel,USB laser b/w	14
SAMSUNG 1200 Parallel,USB laser b/w	15
LEXMARK 600 Parallel,USB laser b/w	16
OKIDATA 1200x600 Parallel,USB,LAN laser color	17
BROTHER 600x600 Parallel,USB laser b/w	18

Table B.15: Sample keyboards for var_{kb}

Component specification	Enumeration
ITRON BLACK PS/2	0
LOGITECH BLACK USB,PS/2 kb+mouse wireless	1
BENQ BLACK	2
MICROSOFT IVORY kb+mouse	3
EAGLE TOUCH SILVER+BLACK USB,PS/2	4
ELUMINX BLACK+BLUE PS/2	5
IONE BLACK USB,PS/2	6
ZIPPY SILVER+BLUE USB	7
LOGITECH BLACK USB,PS/2	8
TSUNAMI BLACK PS/2 kb+mouse	9

B.1.2 The constraints

Table B.16: PC connection constraints in Formulation I.

Constraint variables	Description
$(var_{cpu},\ var_{mb})$	CPU socket must fit on a motherboard (MB), and fsb should be compatible
$(var_{mb},\ var_{ram})$	memory pins and the slots on a MB have to match; if RAM is a dual RAM, a motherboard must support it
(var_{mb},var_{vga})	if a MB has a video chip onboard, a VGA is optional; if a VGA is to use, the interface must be supported
$(var_{mb},\ var_{snd})$	if a MB has a sound chip, a sound card is optional; if a sound card is to use, the interface must be supported
$(var_{mb},\ var_{nic})$	if a MB has a network chip, an NIC is optional; if an NIC is to use, the interface must be supported
$(var_{mb},\ var_{fdd})$	the connection interface must be supported by a MB
$(var_{mb},\ var_{hdd})$	the connection interface must be supported by a MB
$(var_{mb},\ var_{cd})$	the connection interface must be supported by a MB
$(var_{mb},\ var_{tower})$	a motherboard formation factor must be consistent
$(var_{tower},\ var_{power})$	if the tower case includes a power supply, an additional power supply is optional
$(var_{vga},\ var_{scr})$	a monitor connector must be supported by a VGA
$(var_{prt},\ var_{mb},\ var_{tower})$	a printer connector must be supported by a MB or a tower
$(var_{kb},\ var_{mb},\ var_{tower})$	a KB connector must be supported by a MB or a tower
$(var_{mouse},\ var_{mb},\ var_{tower})$	a mouse connector must be supported by a MB or a tower
$(var_{cpu},\ var_{ram},\ var_{mb},\ var_{vga},\ var_{snd},\ var_{nic},\ var_{fdd},\ var_{hdd},\ var_{cd},\ var_{scr},\ var_{prt},\ var_{kb},\ var_{mouse},\ var_{tower},\ var_{power})$	total price must be smaller than or equal to the budget

B.1.3 Description for Formulation I test problems

B.1.3.1 Problem 10.3

The is the base problem of problem set 10. It has 14 variables and 14 connection constraints among the variables. The domain size of each variable is at least 10 on average and several variables have 30 to 40 samples. A list of constraints is shown in Table B.16. The search space is $1,006,236,000,000,000 \approx 1.01 \times 10^{15}$.

B.1.3.2 Problem 10.40

Based on problem 10.3, an `ext_color` constraint is added and everything else stays the same. The `ext_color` constraint enforces the colors of casing, floppy disk drive, CD-ROM, monitor, keyboard and mouse to be consistent. For example, they must all be:

- ('BLACK', 'BRONZE', 'BLACK/SILVER'), or
- ('SILVER', 'GREY', 'SILVER/BLACK'), or
- ('WHITE', 'IVORY', 'BEIGE'),

each tuple represents a set of compatible colors.

B.1.3.3 Problem 10.41

Based on problem 10.40, this problem takes user's budget into account and adds two 14-ary price constraints. These two price constraints define an upper bound and a lower bound of the budget: `'UPPERprice(items, 1800)'` and `'LOWERprice(items, 1500)'` respectively.

B.1.3.4 Problem 10.53

Based on problem 10.3, we add a price constraint `'UPPERprice(items, 750)'`. The price constraint is harder than those in problem 10.41.

B.1.3.5 Problem 10.55

This problem is designed to compare the result with problem 10.53. The only difference between these two problems is that the search space here is ordered; i.e. each set of domain is arranged in the order of item prices.

B.1.3.6 Problem 10.62

Based on problem 10.3, we add a price constraint `'UPPERprice(items, 500)'`, which makes the problem no solution. This problem test only those algorithms using distance objective function. For this problem, the distance objective function is modified so that the evaluation of the price constraint returns some value smaller than 1 and greater or equal to 0. The test result will be accepted as soon as all constraints are satisfied, except for the price constraint.

B.1.3.7 Problem 10.64

This problem is designed to compare the result with problem 10.62. The only difference between these two problems is that the search space here is ordered; i.e. each set of domain is arranged in the order of item prices. Like problem 10.62, this is to test those algorithms using distance objective function.

B.2 Formulation II

B.2.1 The variables and the domains

Table B.17: Sample values of CPU specifications and the enumerated domain.

Enum	cpu_{brand}	cpu_{model}	cpu_{clock}	cpu_{socket}	cpu_{fsb}
0		ATHLON	1.75	S478	266
1	AMD	Mobile ATHLON	1.8	S754	333
2	INTEL	PENTIUM	1.86	S939	800
3		SEMPRON	2.0	SOCKETA	1600
4			2.2		2000
5			2.8		
6			3.0		
7			3.2		

The rows in the table do not represent a product, but the enumerated values. For example, AMD of cpu_{brand} is 1, ATHLON of cpu_{model} is 0, PENTIUM of cpu_{model} is 2, SOCKETA of cpu_{socket} is 3, etc.

Table B.18: Sample values of RAM specifications and the enumerated domain.

Enum	ram_{brand}	ram_{pin}	ram_{MB}	ram_{dual}
0		184	256	False
1			512	True
2			1024	
3	SAMSUNG			
4	CORSAIR			
5	OCZ			
6	KINGSTON			
7	INFINEON			

Variable ram_{MB} represents the capacity of a RAM. The rows in the table do not represent a product, but the enumerated values. For example, KINGSTON of ram_{brand} is 6, 512MB of ram_{MB} is 1, etc.

Table B.19: Sample values of motherboard specifications and the enumerated domain.

Enum	mb_{brand}	mb_{socket}	mb_{form}	Onboard[1]	mb_{pin}	Drv/Slots/Dual[2]	mb_{fsb}
0		S478	ATX	False	184	False	200
1		S754		True		True	266
2							333
3		SocketA					400
4							533
5							800
6							
7							
8	SOLTEK						
9	ASUS						
10	MSI						
11	ASROCK						

Variable mb_{form} represents the formfactor of a motherboard. "Onboard" includes NIC and SoundCard, "Drv" includes IDE or SATA and "Slots" include AGP, IDE, PCI, USB. The rows in the table do not represent a product, but the enumerated values. For example, ASUS of mb_{brand} is 9, S478 of mb_{socket} is 0, True of Onboard for mb_{snd} and mb_{nic} is 1, etc.

1. Combinations of mb_{snd} and mb_{nic}
2. Combinations of mb_{IDE}, mb_{SATA}, mb_{AGP}, mb_{PCI}, mb_{USB} and mb_{dual}

Table B.20: Sample values of VGA specifications and the enumerated domain.

Enum	vga_{brand}	vga_{model}	vga_{face}	vga_{DVI}	vga_{TV}	vga_{VGA}
0		GEFORCE	AGP	False	False	False
1		RADEON		True	True	True
2			PCI			
10	MSI					
12	ATI					
13	LEADTEK					
14	SAPPHIRE					
15	BFG					

Variable vga_{face} represents the interface of a VGA card, and vga_{DVI}, vga_{TV} and vga_{VGA} are the connectors of a VGA card. The rows in the table do not represent a product, but the enumerated values. For example, ATI of vga_{brand} is 12, AGP of vga_{model} is 0, etc.

Table B.21: Sample values of sound card specifications and the enumerated domain.

Enum	snd_{brand}	snd_{face}
0	Dummy	
2		PCI
16	CREATIVE	
17	CHAINTECH	
18	M-AUDIO	
19	AOPEN	

Variable snd_{face} represents the interface of a sound card. "Dummy sound card" is used when sound card is optional. The rows in the table do not represent a product, but the enumerated values. For example, CREATIVE of snd_{brand} is 16, PCI of snd_{face} is 2, etc.

Table B.22: Sample values of NIC specifications and the enumerated domain.

Enum	nic_{brand}	nic_{face}	$nic_{wireless}$
0	Dummy		False
1			True
2	INTEL	PCI	
3		USB	
4		PCMCIA	
20	D-LINK		
21	SMC		
22	MICRONET		
23	3COM		
24	SURECOM		
25	LINKSYS		

Variable nic_{face} represents the interface of a network card. "Dummy network card" is used when network card is optional. The rows in the table do not represent a product, but the enumerated values. For example, LINKSYS of nic_{brand} is 25, False of $nic_{wireless}$ is 0, etc.

Table B.23: Sample values of floppy drive specifications and the enumerated domain.

Enum	fdd_{brand}	fdd_{color}	fdd_{ext}	fdd_{face}
0			False	IDE
1		BLACK	True	
2		IVORY		USB
9	ASUS			
26	MITSUMI			
27	Generic			
28	SONY			

Variable fdd_{face} represents the interface of a floppy drive, and fdd_{ext} indicates whether a floppy drive is external or not. The rows in the table do not represent a product, but the enumerated values. For example, SONY of fdd_{brand} is 28, IDE of fdd_{face} is 0, etc.

Table B.24: Sample values of hard drive specifications and the enumerated domain.

Enum	hdd_{brand}	hdd_{MB}	hdd_{ext}	hdd_{face}	hdd_{rpm}
0		36	False	IDE	7200
1		74	True	SATA	10000
2		80			
3		120			
4		200			
29	WD				
30	SEAGATE				
31	MAXTOR				

Variable hdd_{face} represents the interface of a hard drive and hdd_{MB} is the capacity of a hard drive in Megabyte. The rows in the table do not represent a product, but the enumerated values. For example, SEAGATE of hdd_{brand} is 30, 120MB of hdd_{MB} is 3, etc.

Table B.25: Sample values of CD-ROM specifications and the enumerated domain.

Enum	cd_{brand}	cd_{color}	cd_{ext}	cd_{face}	cd_{wrt}
0			False	IDE	False
1		BLACK	True		True
2		IVORY			
3		BEIGE			
4		GREY			
5		WHITE			
9	ASUS				
10	MSI				
19	AOPEN				
28	SONY				
32	LG				
33	LITEON				
34	BENQ				
35	TOSHIBA				

Variable cd_{face} represents the interface of a CD-ROM drive and cd_{wrt} indicates whether the CD-ROM drive is a writer or not. The rows in the table do not represent a product, but the enumerated values. For example, TOSHIBA of cd_{brand} is 35, BLACK of cd_{color} is 1, True of cd_{wrt} is 1, etc.

Table B.26: Sample values of power supply specifications and the enumerated domain.

Enum	$power_{brand}$	$power_{watts}$
0	DUMMY	0
1		300
2		350
3		420
4		430
5	OCZ	480
6		500
7		520
8		550
39	THERMALTAKE	
40	ANTEC	
41	ENERMAX	
42	SPARKLE	
43	generic	
44	ULTRA	

"Dummy power supply" is used when power supply is optional. The rows in the table do not represent a product, but the enumerated values. For example, ANTEC of $power_{brand}$ is 40, 340w of $power_{watts}$ is 2, etc.

Table B.27: Sample values of tower case specifications and the enumerated domain.

Enum	$tower_{brand}$	$tower_{form}$	$tower_{power}$	$tower_{PS}$	$tower_{USB}$	$tower_{watts}$	$tower_{color}$
0		ATX	None	False	False	0	
1		MicroATX	ATX	True	True	300	BLACK
2						350	IVORY
3						380	BEIGE
4						400	
5						420	
6							
7							SILVER
8							S/B
9							BLUE
10							GREEN
11							YELLOW
12							B/S
13							BRONZE
40	ANTEC						
45	TSUNAMI						
46	RAIDMAX						
47	ASPIRE						
48	NGEAR						

The rows in the table do not represent a product, but the enumerated values. For example, NGEAR of $tower_{brand}$ is 48, SILVER of $tower_{color}$ is 7, etc. Color 'S/B' means silver/black and 'B/S' means black/silver

B.2.2 The constraints

B.2.2.1 Component constraints

Table B.28: Sample CPUs in good tuples–component constraint "GOODcpu".

cpu	cpu_{brand}	cpu_{model}	cpu_{clock}	cpu_{socket}	cpu_{fsb}
(0,	1,	0,	3,	1,	2)
(1,	1,	1,	2,	3,	0)
(2,	2,	2,	6,	0,	2)
(3,	1,	0,	4,	1,	2)
(4,	1,	0,	4,	2,	4)
(5,	1,	3,	0,	3,	1)
(6,	2,	2,	5,	0,	2)
(7,	2,	2,	6,	0,	2)
(8,	1,	0,	1,	1,	2)
(9,	2,	2,	7,	0,	2)

We represent "AMD ATHLON 64 3000+ 2.0GHz S754 800fsb" in (1, 0, 3, 1, 2) for brand, model, clock, socket and fsb.

Table B.29: Sample RAMs in good tuples–component constraint "GOODram".

RAM#	Specification
0	(3, 0, 1, 0)
1	(4, 0, 2, 1)
2	(5, 0, 1, 0)
3	(4, 0, 2, 1)
4	(3, 0, 0, 0)
5	(6, 0, 2, 1)
6	(5, 0, 1, 1)
7	(5, 0, 2, 1)
8	(7, 0, 1, 0)
9	(6, 0, 1, 1)

Table B.30: Sample motherboards in good tuples–component constraint "GOODmb".

Motherboard#	Specification
0	(8, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 3)
	(8, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 2)
	(8, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1)
1	(9, 3, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 3)
2	(9, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 5)
3	(9, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 5)
	(9, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 4)
	(9, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 3)
4	(8, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 5)
5	(10, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 5)
6	(9, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 5)
	(9, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 4)
	(9, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 3)
7	(9, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 5)
8	(11, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 2)
	(11, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1)
	(11, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0)
9	(9, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 3)
	(9, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 2)
	(9, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1)
	(9, 3, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0)

The sample motherboards (mb) in order of brand, socket, form factor, NIC, sound card, pin, drv_IDE, drv_SATA, AGP, IDE, PCI, USB, dual type and fsb are the following. Because of the multiple supported fsb, there can be multiple entries for some motherboard.

Table B.31: Sample VGAs in good tuples–component constraint "GOODvga".

Video#	Specification
0	(12, 1, 0, 1, 1, 1)
1	(10, 1, 0, 1, 1, 1)
2	(13, 0, 0, 1, 1, 1)
3	(12, 1, 0, 1, 1, 1)
4	(13, 0, 0, 1, 1, 1)
5	(12, 1, 0, 1, 1, 1)
6	(14, 1, 0, 1, 1, 1)
7	(10, 0, 0, 1, 1, 0)
8	(14, 0, 0, 1, 1, 1)
9	(15, 0, 0, 1, 1, 1)

Table B.32: Sample sound cards in good tuples–component constraint "GOODsnd".

Sound#	Specification
0	(0, 2)
1	(16, 2)
2	(16, 2)
3	(16, 2)
4	(16, 2)
5	(17, 2)
6	(18, 2)
7	(16, 2)
8	(19, 2)
9	(18, 2)
10	(16, 2)

Table B.33: Sample network cards in good tuples–component constraint "GOODnic".

NIC#	Specification
0	(0, 2, 0)
1	(20, 2, 0)
2	(21, 2, 0)
3	(22, 2, 0)
4	(2, 2, 0)
5	(20, 4, 0)
6	(2, 2, 0)
7	(23, 2, 0)
8	(24, 2, 0)
9	(25, 3, 0)
10	(25, 2, 0)

Table B.34: Sample floppy drives in good tuples–component constraint "GOODfdd".

FDD#	Specification
0	(26, 1, 0, 0)
1	(26, 2, 0, 0)
2	(27, 1, 0, 0)
3	(27, 2, 0, 0)
4	(28, 1, 0, 0)
5	(28, 1, 1, 3)
6	(9, 2, 1, 3)

Table B.35: Sample hard drives in good tuples–component constraint "GOODhdd".

HDD#	Specification
0	(29, 4, 0, 0, 0)
1	(30, 4, 0, 0, 0)
2	(30, 4, 0, 1, 0)
3	(30, 2, 0, 0, 0)
4	(30, 3, 0, 0, 0)
	(30, 3, 0, 1, 0)
5	(30, 3, 0, 0, 0)
6	(31, 2, 0, 0, 0)
7	(29, 1, 0, 1, 1)
8	(29, 0, 0, 1, 1)
9	(29, 2, 0, 0, 0)

Table B.36: Sample CD-ROM drives in good tuples–component constraint "GOODcd".

CD#	Specification
0	(32, 2, 0, 1, 0)
1	(33, 1, 0, 1, 0)
2	(32, 1, 0, 1, 0)
3	(24, 3, 0, 1, 0)
4	(9, 2, 0, 1, 0)
5	(28, 3, 0, 1, 0)
6	(9, 4, 0, 1, 0)
7	(33, 2, 0, 1, 0)
8	(10, 2, 0, 1, 0)
9	(19, 5, 0, 1, 0)
10	(32, 5, 0, 1, 1)
11	(33, 2, 0, 1, 1)
12	(32, 1, 0, 1, 1)
13	(35, 3, 0, 1, 1)
14	(32, 1, 0, 1, 1)
15	(34, 3, 0, 1, 1)
16	(32, 2, 0, 1, 1)
17	(33, 1, 0, 1, 1)
18	(28, 5, 0, 1, 1)
19	(33, 3, 0, 1, 1)

Table B.37: Sample power supplies in good tuples–component constraint "GOODpower".

Power#	Specification
0	(0, 0)
1	(39, 3)
2	(5, 7)
3	(40, 5)
4	(41, 2)
5	(42, 1)
6	(43, 2)
7	(40, 4)
8	(40, 5)
9	(44, 6)
10	(40, 8)

Table B.38: Sample tower cases in good tuples–component constraint "GOODtower".

Casing#	Specification
0	(40, 0, 1, 1, 1, 3, 1)
1	(40, 0, 1, 1, 1, 2, 1)
2	(45, 0, 1, 1, 1, 4, 2)
3	(46, 0, 1, 1, 1, 5, 1)
4	(40, 0, 0, 1, 1, 0, 7)
5	(40, 1, 1, 1, 1, 1, 8)
6	(47, 0, 1, 1, 1, 2, 1)
	(47, 0, 1, 1, 1, 2, 7)
	(47, 0, 1, 1, 1, 2, 9)
	(47, 0, 1, 1, 1, 2, 10)
	(47, 0, 1, 1, 1, 2, 11)
7	(48, 0, 1, 1, 1, 2, 12)
8	(40, 0, 1, 1, 1, 1, 13)
9	(48, 0, 1, 1, 1, 2, 3)

B.2.2.2 Connection constraints

Table B.39: Sample PC connection constraints in Formulation II.

<table>
<tr><th>Constraint expression</th><th>Description</th></tr>
<tr><td>CPU_socket == MB_socket</td><td>CPU socket must fit on a motherboard</td></tr>
<tr><td>CPU_fdb == MB_fsb</td><td>CPU fsb should be compatible</td></tr>
<tr><td>RAM_pin == MB_pin</td><td>Memory pins and the slots on a MB have to match</td></tr>
<tr><td>RAM_dual <= MB_dual</td><td>If RAM is dual, a motherboard must support it</td></tr>
<tr><td>VGA_face < 4 and
(((VGA_face+1)
* (MB_AGP + MB_PCI*100))
/ 10**VGA_face) % 10) > 0</td><td>If a VGA is used, the interface must be supported</td></tr>
<tr><td>(SND_brand != 0) | (MB_snd != 0)</td><td>If a MB has a sound chip, a sound card is optional</td></tr>
<tr><td>SND_face < 4 and
(((SND_face+1) * (MB_PCI*100))
/ 10**SND_face) % 10) > 0</td><td>If sound card is used, interface must be supported</td></tr>
<tr><td>(NIC_brand != 0) | (MB_nic != 0)</td><td>If a MB has a network chip, an NIC is optional</td></tr>
<tr><td>NIC_face < 4 and
(((NIC_face+1) * (MB_PCI*100))
/ 10**NIC_face) % 10) > 0</td><td>If an NIC is used, the interface must be supported</td></tr>
<tr><td>FDD_face < 3 and
(((FDD_face+1) * (MB_drvIDE
+ MB_IDE + MB_USB*100))
/ 10**FDD_face) % 10) > 0</td><td>The interface must be supported by a MB</td></tr>
<tr><td>HDD_face < 3 and
(((HDD_face+1) * (MB_drvIDE
+ MB_IDE + MB_drvSATA*10
+ MB_USB*100)) / 10**HDD_face)
% 10) > 0</td><td>The interface must be supported by a MB</td></tr>
<tr><td>CD_face < 3 and
(((CD_face+1) * (MB_drvIDE + MB_IDE
+ MB_drvSATA*10 + MB_USB*100))
/ 10**CD_face) % 10) > 0</td><td>The interface must be supported by a MB</td></tr>
<tr><td>(TOWER_power != 0)
| (POWER_brand != 0)</td><td>If the tower case includes a power supply,
an additional power supply is optional</td></tr>
</table>

B.2.2.3 User constraints

Table B.40: Sample PC user constraints in Formulation II.

Constraint expression	Description
`UPPERprice(1800)`	budget upper bound $1800
`LOWERprice(1500)`	price lower bound $1500
`FDD_external == 1`	want to have an external floppy drive
`CD_writer != 1`	do not want a DVD writer

B.2.3 Description for Formulation II test problems

B.2.3.1 Problem set 20 – problem 20.3.

These problems are the simplest set of problems in Formulation II. Problem 20.3 is a base problem that includes 32 variables, 8 component constraints and 12 arithmetic connections. The variables are listed in Table B.41 and the size of the domains varies from 2 to 20. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39. The search space is $98,099,527,680,000,000 \approx 9.81 \times 10^{16}$.

Table B.41: CSP variables of problem set 20.

Component variables	Specification variables
cpu	cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	$vga_{interface}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{watts}$

B.2.3.2 Problem set 21 – problem 21.3.

This set of problems are composed of 36 variables, 8 component constraints and 12 arithmetic connections. The variables are listed in Table B.42 and the sizes of the domains are between 2 to 20. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39. The size of the search space is $5,885,971,660,800,000,000 \approx 5.89 \times 10^{18}$.

Table B.42: CSP variables of problem set 21.

Component variables	Specification variables
cpu	cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{model}, $vga_{interface}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$, $hdd_{external}$, hdd_{mb}
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{watts}$

B.2.3.3 Problem set 22 – problem 22.3.

The size of this set of problems becomes 46, and the base problem 22.3 contains 9 component constraints and 14 arithmetic connections. The variables are listed in Table B.43 and the sizes of the domains are between 2 to 20. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39. Its search space is $361,634,098,839,552,000,000,000 \approx 3.62 \times 10^{23}$.

Table B.43: CSP variables of problem set 22.

Component variables	Specification variables
cpu	cpu_{model}, cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{model}, $vga_{interface}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$, $hdd_{external}$, hdd_{mb}, hdd_{cache}, hdd_{rpm}
cd	cd_{color}, $cd_{interface}$, $cd_{external}$, cd_{writer}
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{PS/2}$, $tower_{USB}$, $tower_{watts}$

B.2.3.4 Problem set 23 – problem 23.3.

The problem size is 51 and the variables are listed in Table B.44. The number of constraints of the base problem 23.3 is the same as problem 22.3. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39. The search space is $11,572,291,162,865,664,000,000,000 \approx 1.16 \times 10^{25}$.

Table B.44: CSP variables of problem set 23.

Component variables	Specification variables
cpu	cpu_{model}, cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, $mb_{sndonboard}$, $mb_{niconboard}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{model}, $vga_{interface}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$, $hdd_{external}$, hdd_{mb}, hdd_{cache}, hdd_{rpm}
cd	cd_{color}, $cd_{interface}$, $cd_{external}$, cd_{writer}
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{PS/2}$, $tower_{USB}$, $tower_{watts}$

B.2.3.5 Problem set 24 – problem 24.3.

The base problem consists of 58 variables, 11 component constraints and 20 arithmetic connections. The variables are listed in Table B.45, sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39. The space becomes 336, 059, 335, 369, 618, 882, 560, 000, 000, 000 $\approx 3.36 \times 10^{29}$.

Table B.45: CSP variables of problem set 24.

Component variables	Specification variables
cpu	cpu_{model}, cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, $mb_{sndonboard}$, $mb_{niconboard}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{model}, $vga_{interface}$, vga_{DVI}, vga_{TV}, vga_{VGA}
snd	snd_{brand}, $snd_{interface}$
nic	nic_{brand}, $nic_{interface}$, $nic_{wireless}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$, $hdd_{external}$, hdd_{mb}, hdd_{cache}, hdd_{rpm}
cd	cd_{color}, $cd_{interface}$, $cd_{external}$, cd_{writer}
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{PS/2}$, $tower_{USB}$, $tower_{watts}$

B.2.3.6 Problem set 25 – problem 25.3.

There are 59 variables for this set of problems and they are listed in Table B.46. Its search space is slightly increased to 672, 118, 670, 739, 237, 765, 120, 000, 000, 000 $\approx 6.72 \times 10^{29}$. The base problem 25.3 contains 11 component constraints and 20 arithmetic connections. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39.

B.2.3.6.1 Problem 25.43. Based on the base problem 25.3, several user constraints are added to the problem: `ramMB`, `fddExt`, `cdDvd` and `cdWriter`.

- `ramMB > 512MB` defines the size of the RAM to be greater than or equal to 512MB.

Table B.46: CSP variables of problem set 25.

Component variables	Specification variables
cpu	cpu_{model}, cpu_{socket}, cpu_{fsb}
ram	ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, $mb_{sndonboard}$, $mb_{niconboard}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{model}, $vga_{interface}$, vga_{DVI}, vga_{TV}, vga_{VGA}
snd	snd_{brand}, $snd_{interface}$
nic	nic_{brand}, $nic_{interface}$, $nic_{wireless}$
fdd	fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	$hdd_{interface}$, $hdd_{external}$, hdd_{mb}, hdd_{cache}, hdd_{rpm}
cd	cd_{color}, $cd_{interface}$, $cd_{external}$, cd_{writer}, cd_{dvd}
$power$	$power_{watts}$
$tower$	$tower_{color}$, $tower_{form}$, $tower_{PS/2}$, $tower_{USB}$, $tower_{watts}$

- `fddExt` $fdd_{external}$!= $hdd_{external}$ == $cd_{external}$ for that if floppy disk drive is an external model, hard drive and CD-ROM must be internal; or vice versa.
- `cdDvd = 1` demands a DVD driver, rather than a CD-ROM.
- `cdWriter = 1` demands only a writer, not a reader.

B.2.3.6.2 Problem 25.47. In addition to the constraints added to problem 25.43, some additional user constraints are added to this problem: `hddCapacity`, `vgaModel`, `onboard` and `ext_color`.

- `hddCapacity > 120GB` defines the capacity of the hard drive to be greater than or equal to 120GB.
- `vgaModel =` 'RADEON' requires the video card model to be 'RADEON'.
- `onboard` ($mb_{sndonboard}$ == $mb_{niconboard}$ == 0) | ($mb_{sndonboard}$!= 0 != $mb_{niconboard}$) defines either both sound card and network card are built in on motherboard or neither of them built in on motherboard.
- `ext_color` enforces the colors of casing, floppy disk drive and CD-ROM to be consistent. For example, they must all be (`'BLACK'`, `'BRONZE'`, `'BLACK/SILVER'`), or

('SILVER', 'GREY', 'SILVER/BLACK'), or ('WHITE', 'IVORY', 'BEIGE'), and each tuple represents compatible colors.

B.2.3.6.3 Problem 25.50. This problem is built on top of problem 25.47. A few user constraints are added: `powerWatts`, `colorBlack` and the price constraints `UPPERprice` and `LOWERprice`.

- `powerWatts >= 350 | towerWatts >= 350` requires the power of the power supplies must be greater than or equal to 350 watts.
- `colorBlack` (fdd_{color} != $tower_{color}$ == cd_{color}) requests the casing and the CD-ROM must be the same color, but the color of the floppy drive must be different.
- `UPPERprice <= 1800` defines price constraint upper bound.
- `LOWERprice >= 1500` defines price constraint lower bound.

B.2.3.6.4 Problem 25.53. Similar to problem 10.53 in problem set 10 (Formulation I), this problem is defined to be harder than problem 25.50 with a harder price constraint upper bound `'UPPERprice(items, 750)'`. The variables and the other constraints are exact same as problem 25.3.

B.2.3.6.5 Problem 25.55. This problem is designed to compare the result with problem 25.53. The only difference is the search space of this problem is ordered; i.e. each set of domain is arranged in the order of item price.

B.2.3.6.6 Problem 25.62. Based on the base problem 25.3, an additional price constraint `'UPPERprice(items, 500)'` is added to make the problem no solution. This problem is meant to test those algorithms with distance objective function. For this problem, the distance objective function is modified in a way that the evaluation of the price constraint returns some value smaller than 1 and greater or equal to 0. The test result will be accepted as soon as all constraints are satisfied, except for the price constraint.

B.2.3.6.7 Problem 25.64. This problem is designed to compare the result with problem 25.62. The only difference is the search space of this problem is ordered; i.e. each set of domain is arranged in the order of item price. Like problem 25.62, this is to test those algorithms with distance objective function.

B.2.3.7 Problem set 26 – problem 26.3.

The problem size is promoted to 70. The variables are listed in Table B.47. There are 11 component constraints and 20 arithmetic connections binding the 70 variables. The search space jumps to $11,356,117,060,810,161,279,467,520,000,000,000,000 \approx 1.14 \times 10^{37}$. Sample component constraints are listed in Table B.28 to Table B.38, and connection constraints are included in Table B.39.

Table B.47: CSP variables of problem set 26.

Component variables	Specification variables
cpu	cpu_{brand}, cpu_{model}, cpu_{socket}, cpu_{clock}, cpu_{fsb}
ram	ram_{brand}, ram_{pin}, ram_{mb}, ram_{dual}
mb	mb_{brand}, mb_{socket}, mb_{drvIDE}, $mb_{drvSATA}$, $mb_{sndonboard}$, $mb_{niconboard}$, mb_{form}, mb_{pin}, mb_{AGP}, mb_{IDE}, mb_{PCI}, mb_{USB}, mb_{dual}, mb_{fsb}
vga	vga_{brand}, vga_{model}, $vga_{interface}$, vga_{DVI}, vga_{TV}, vga_{VGA}, vga_{memory}
snd	snd_{brand}, $snd_{interface}$
nic	nic_{brand}, $nic_{interface}$, $nic_{wireless}$
fdd	fdd_{brand}, fdd_{color}, $fdd_{interface}$, $fdd_{external}$
hdd	hdd_{brand}, $hdd_{interface}$, $hdd_{external}$, hdd_{mb}, hdd_{cache}, hdd_{rpm}
cd	cd_{brand}, cd_{color}, $cd_{interface}$, $cd_{external}$, cd_{writer}, cd_{dvd}
$power$	$power_{brand}$, $power_{watts}$
$tower$	$tower_{brand}$, $tower_{color}$, $tower_{form}$, $tower_{PS}$, $tower_{USB}$, $tower_{watts}$

Appendix C

Experimental Setup and Evaluation Data

C.1 Parameter Settings for Exploration Phase

Table C.1: Parameter settings used in Exploration phase

model/objective	parameter	description	values
(across algorithm)	*pop*	the population of the swarm	20, 50, 100
	k	the size of neighbourhood	6, global only[1]
	ITER	the maximum of iterations	10000
Continuous/conflict	ω	an inertia weight in computing particle's velocity $v(t)$, determines the effect of $v(t-1)$	decreases from 0.9 to 0.4
	(c_1, c_2)	the acceleration constants in computing particle's velocity, determines the influence of the global (or local) best and the individual best information	(2,2), (3,1), (1,3)
	nohope	an iteration count, defines when the swarm has no more improvement for so long, the swarm performs a certain strategy to break the situation	500 (or 1000)[2]

model/objective	parameter	description	values
	pop_rate	the percentage of particles to perform the given strategy after the *nohope* count kicks in; for instance, *pop_rate* = 0.5 means half of the population should perform the specific strategy	0.25,0.375,0.5,0.625,0.75
	dfs_size	defines the number of variables in each depth-first search group assigned to a particle; see Section 4.2.2.8	3
	regroup	an iteration count, defines when the swarm should perform such a strategy; only used in algorithms involving "exchange partner" strategy	200, 500, 1000
	stop_group	an iteration count, defines when to stop regrouping particles and return to normal	3333, 5000, 10000
	spawn	an iteration count, defines when to spawn more particles; used in genericHybrid algorithm	$2 \times$ *nohope*
Discrete/conflict	ϕ_1 and ϕ_2	the acceleration constants in computing particle's velocity	random(0,4) and $4 - \phi_1$
	(v_{max}, v_{min})	the velocity upper bound and lower bound in determining particle's velocity	(4,-4), (2,-2), (1,-1)
	nohope	same as in the Continuous	500
	pop_rate	same as in the Continuous	0.25,0.375,0.5,0.625,0.75
BCSP/conflict	φ_1, φ_2	the coefficients used in computing particle's velocity	{0, 1}
	$deflection$	serves as a switch to refine particle's moving direction, i.e. whether a particle should flip the direction or not	0, $1/n$, $2/n$
	pop_rate	same as in the Continuous	0.1, 0.15, 0.2, 0.25, 0.3

model/objective	parameter	description	values
	nohope	exists in the original BCSP model for individual particles to determine when it has done no improvement and should restart; we also use this parameter globally to the swarm similar to the *nohope* in the Continuous model	500
Continuous/distance		same as Continuous/conflict	
Discrete/distance		same as Discrete/conflict	

1. "Global only" means the entire swarm as one and the only one neighbourhood.
2. The *nohope* count is set depending on the maximum number of iterations (i.e. 1/20 of *ITER*); in turns, 500 is used in *genericHop* and *genericRestart*, and 1000 is used in *genericMultigbest*, *genericDFS* and *zigzagDFS*.

C.2 Parameter Settings for Comparison Phase

Table C.2: Parameter settings used in Comparison phase

model/objective	parameter	description	values
(across algorithm)	*pop* *k* *ITER*	the population of the swarm the size of neighbourhood the maximum of iterations	3, 5, 10 2 20000 (or 50000)[1]
Continuous/conflict	ω	an inertia weight in computing particle's velocity, determines the effect of the previous velocity at time $t-1$	decreases from 0.9 to 0.4
	(c_1, c_2)	the acceleration constants in computing particle's velocity, determines the influence of the global (or local) best information and the individual best information	(3, 1)

model/objective	parameter	description	values
	pop_rate	the percentage of particles to perform the given strategy after the *nohope* count kicks in; for instance, *pop_rate* = 0.5 means half of the population should perform the specific strategy	0.25, 0.5, 0.75
	nohope	an iteration count, defines when the swarm has no more improvement for so long, the swarm performs a certain strategy to break the situation	see note[2]
	dfs_size	defines the number of variables in each depth-first search group assigned to a particle; see Section 4.2.2.8	3
	regroup	an iteration count, defines when the swarm should perform such a strategy; only used in algorithms involving "exchange partner" strategy	200, 500, 1000
	stop_group	an iteration count, defines when to stop regrouping particles and return to normal	66667, 10000, 20000
	spawn	an iteration count, defines when to spawn more particles; used in genericHybrid algorithm	$2 \times nohope$
Discrete/conflict	ϕ_1 and ϕ_2	the acceleration constants in computing particle's velocity	random(0,4) and $4 - \phi_1$
	(v_{max},v_{min})	the velocity upper bound and lower bound in determining particle's velocity	(2.0 ,-2.0)
	nohope	same as in the Continuous	see note[2]
	pop_rate	same as in the Continuous	0.25, 0.5, 0.75
BCSP/conflict	(φ_1, φ_2)	the coefficients used in computing particle's velocity	(0, 0)

model/objective	parameter	description	values
	deflection	serves as a switch to refine particle's moving direction, i.e. whether a particle should flip the direction or not	0, $1/n$, $2/n$
	nohope	exists in the original BCSP model for individual particles to determine when it has done no improvement and should restart; we also use this parameter globally to the swarm similar to the *nohope* in the Continuous model	see note[2]
	pop_rate	same as in the Continuous	0.1, 0.2, 0.3
Continuous/distance		same as Continuous/conflict	
Discrete/distance		same as Discrete/conflict	

1. Because zigzag and zigzagHop type algorithms only process one dimension per iteration, it can generally do faster than other type algorithms. Thus, given approximately same amount of run time, we can assign a much higher iteration upper bound to these algorithms. For consistent, we define $ITER = 50000$ for zigzag and zigzagHop type algorithms, and 20000 is for all the other algorithms.

2. The *nohope* count is set depending on the maximum number of iterations (i.e. 1/20 of $ITER$); in turns, 1000 is used in *genericHop* and *genericHybrid*, 2500 is used in *genericZigzagHop*, and 5000 is used *zigzagDFS* algorithm.

C.3 Figures

Table C.3: The *partial* success rate of PSO algorithms: problem set 10 from the Comparison phase.

The success rate of PSO algorithms					
PSO Algorithm	Continuous conflict	Discrete conflict	BCSP conflict	Continuous distance	Discrete distance
genericPSO	43.3%	-	-	47.2%	-
genericZigzag	45%	-	-	49.4%	-
genericHop	56.4%	-	-	64.1%	-
genericZigzagHop	60.3%	-	-	58.1%	-
genericExchange	67.1%	-	-	55.8%	-
zigzagExchange	65.8%	-	-	59.6%	-
zigzagDFS	52.3%	-	-	50%	-
genericHybrid	67.1%	-	-	69.5%	-
binaryDiscrete	-	59%	-	-	56.1%
binaryZigzag	-	60%	-	-	61.3%
binaryHop	-	60%	-	-	68.2%
binaryZigzagHop	-	60%	-	-	65.7%
grayDiscrete	-	58%	-	-	58.3%
grayZigzag	-	60%	-	-	61.9%
grayHop	-	60%	-	-	48.7%
bcspPSO	-	-	26.1%	-	-
bcspZigzag	-	-	51.3%	-	-
bcspHop	-	-	60%	-	-
bcspZigzagHop	-	-	60%	-	-

These figures are for reference only. The experiments with problem 10.53~10.64 are *incomplete*, not including *pop* = 10.

Table C.4: The success rate of the PSO parameter settings.

Algorithm and settings					Problem					
pop	*pop_rate*	*regroup*	*stop_group*	*deflection*	25.3	25.43	25.47	25.50	25.53	25.55
genericHop of the Continuous-conflict model										
3	0.25	-	-	-	70%	50%	60%	30%	0%	0%
3	0.50	-	-	-	80%	90%	90%	50%	0%	10%
3	0.75	-	-	-	100%	100%	100%	70%	10%	0%
5	0.25	-	-	-	100%	80%	40%	40%	0%	0%
5	0.50	-	-	-	100%	100%	80%	70%	10%	30%

Algorithm and settings					Problem					
pop	*pop_rate*	*regroup*	*stop_group*	*deflection*	25.3	25.43	25.47	25.50	25.53	25.55
5	0.75	-	-	-	100%	100%	100%	70%	20%	30%
10	0.25	-	-	-	90%	90%	90%	70%	0%	10%
10	0.50	-	-	-	100%	100%	100%	100%	0%	30%
10	0.75	-	-	-	100%	100%	100%	90%	10%	20%
genericZigzagHop of the Continuous-conflict model										
3	0.25	-	-	-	100%	100%	90%	50%	0%	10%
3	0.50	-	-	-	100%	100%	90%	60%	10%	0%
3	0.75	-	-	-	100%	100%	100%	60%	10%	10%
5	0.25	-	-	-	100%	90%	90%	70%	0%	0%
5	0.50	-	-	-	100%	100%	100%	90%	0%	30%
5	0.75	-	-	-	90%	100%	100%	80%	10%	10%
10	0.25	-	-	-	100%	100%	100%	100%	10%	40%
10	0.50	-	-	-	100%	100%	100%	100%	10%	50%
10	0.75	-	-	-	100%	100%	100%	90%	0%	40%
genericHybrid of the Continuous-conflict model										
5	0.2	500	10000	-	80%	100%	60%	10%	0%	0%
5	0.2	500	20000	-	90%	80%	60%	40%	0%	10%
5	0.2	1000	10000	-	60%	60%	50%	20%	0%	0%
5	0.2	1000	20000	-	100%	80%	60%	10%	0%	10%
10	0.2	500	10000	-	90%	90%	80%	50%	0%	20%
10	0.2	500	20000	-	100%	90%	90%	20%	0%	0%
10	0.2	1000	10000	-	80%	90%	60%	60%	0%	30%
10	0.2	1000	20000	-	100%	90%	90%	50%	0%	0%
bcspHop of the BCSP model										
3	0.50	-	-	0.0	70%	40%	20%	20%	0%	0%
3	0.75	-	-	0.0	100%	60%	20%	30%	0%	0%
3	0.50	-	-	$1/n$	50%	80%	50%	20%	0%	0%
3	0.75	-	-	$1/n$	90%	90%	80%	60%	0%	0%
3	0.50	-	-	$2/n$	70%	80%	30%	40%	0%	0%
3	0.75	-	-	$2/n$	90%	100%	80%	90%	0%	0%
5	0.50	-	-	0.0	80%	60%	70%	30%	0%	0%
5	0.75	-	-	0.0	100%	100%	90%	50%	0%	10%
5	0.50	-	-	$1/n$	90%	90%	60%	60%	0%	0%
5	0.75	-	-	$1/n$	100%	100%	90%	90%	10%	0%

Algorithm and settings					Problem					
pop	*pop_rate*	*regroup*	*stop_group*	*deflection*	25.3	25.43	25.47	25.50	25.53	25.55
5	0.50	-	-	$2/n$	80%	100%	70%	80%	0%	0%
5	0.75	-	-	$2/n$	100%	90%	80%	80%	0%	0%
10	0.50	-	-	0.0	90%	90%	80%	70%	0%	0%
10	0.75	-	-	0.0	100%	100%	100%	80%	0%	10%
10	0.50	-	-	$1/n$	100%	90%	100%	80%	10%	0%
10	0.75	-	-	$1/n$	100%	100%	100%	100%	0%	0%
10	0.50	-	-	$2/n$	100%	100%	100%	90%	10%	0%
10	0.75	-	-	$2/n$	100%	100%	100%	100%	0%	0%
bcspZigzagHop of the BCSP model										
3	0.50	-	-	0.0	80%	100%	60%	80%	0%	10%
3	0.75	-	-	0.0	90%	100%	50%	60%	0%	10%
3	0.50	-	-	$1/n$	100%	100%	100%	100%	0%	0%
3	0.75	-	-	$1/n$	100%	100%	100%	100%	10%	0%
3	0.50	-	-	$2/n$	100%	100%	100%	100%	10%	0%
3	0.75	-	-	$2/n$	100%	100%	100%	100%	0%	10%
5	0.50	-	-	0.0	100%	100%	50%	40%	20%	0%
5	0.75	-	-	0.0	100%	100%	40%	60%	0%	0%
5	0.50	-	-	$1/n$	100%	100%	100%	100%	0%	0%
5	0.75	-	-	$1/n$	100%	100%	100%	100%	10%	0%
5	0.50	-	-	$2/n$	100%	100%	100%	100%	0%	0%
5	0.75	-	-	$2/n$	100%	100%	100%	100%	10%	0%
10	0.50	-	-	0.0	100%	90%	60%	70%	0%	10%
10	0.75	-	-	0.0	100%	90%	60%	60%	0%	10%
10	0.50	-	-	$1/n$	100%	100%	100%	100%	0%	10%
10	0.75	-	-	$1/n$	100%	100%	100%	100%	20%	20%
10	0.50	-	-	$2/n$	100%	100%	100%	100%	0%	0%
10	0.75	-	-	$2/n$	100%	100%	100%	100%	20%	40%

The outcomes are the individual PC configuration problem set 25 from the Comparison phase.

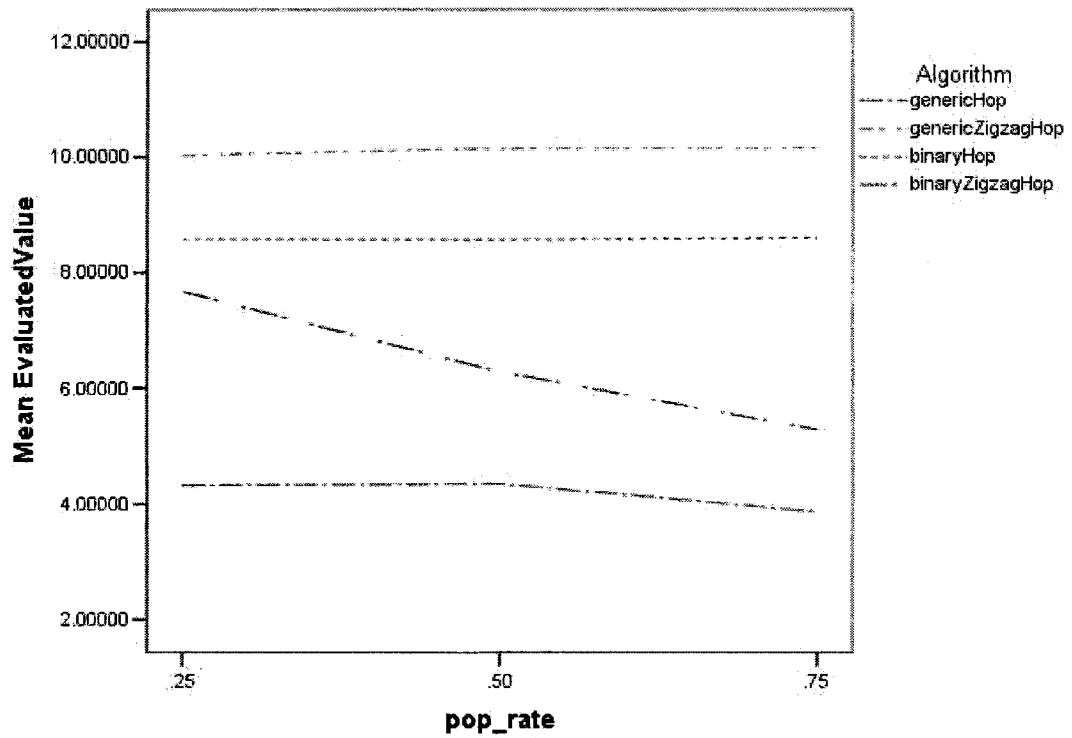
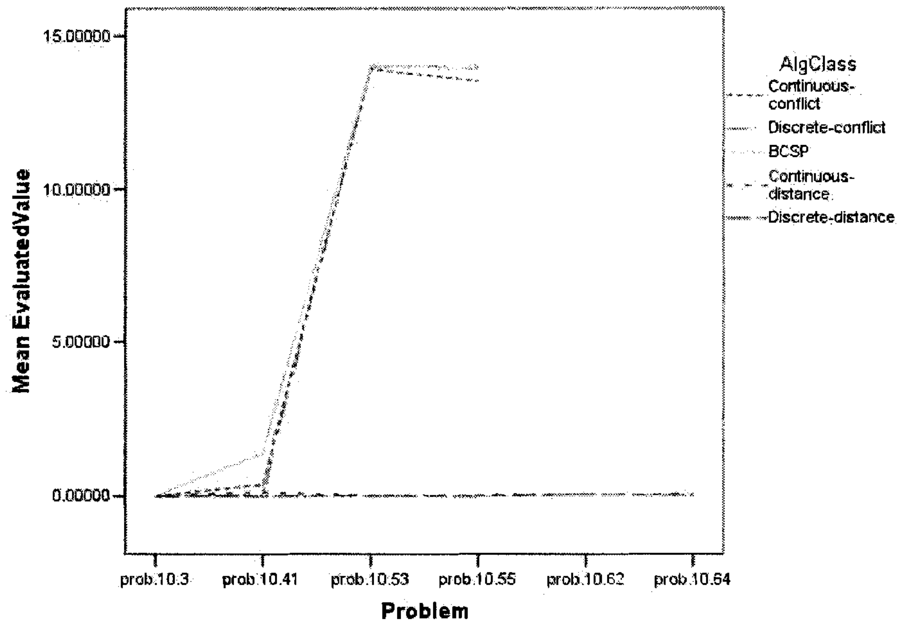
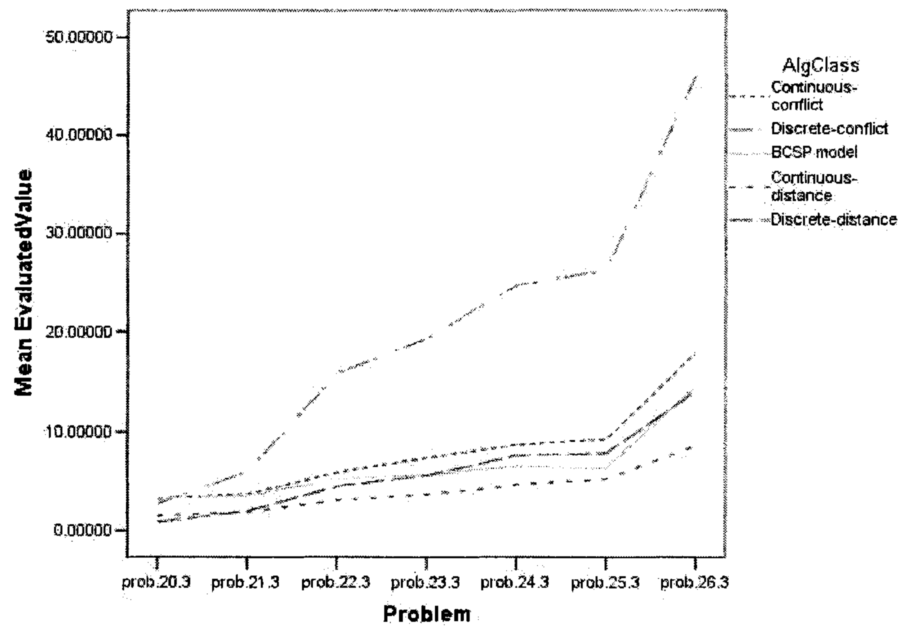


Figure C.1: The mean evaluation value of *pop_rate*: problem sets 25 from the Comparison phase.

This includes the hop and the zigzagHop type algorithms using the distance function and the outcomes include all PC configuration problem set 25 from the Comparison phase. See Figure 5.6 for the algorithms using the conflict count function. Generally, the higher the *pop_rate*, the lower the mean evaluation value.



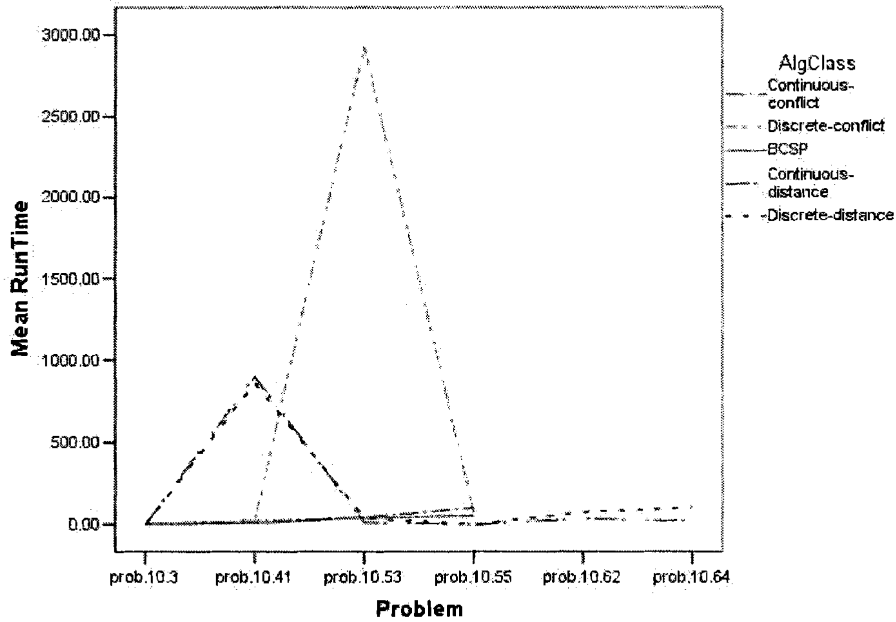
(a) Problem set 10



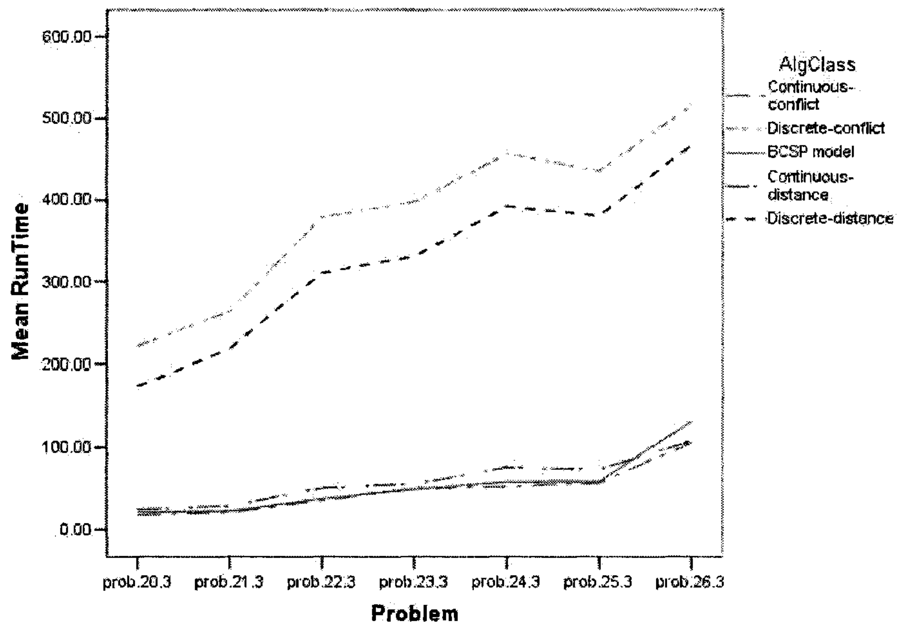
(b) Problems 20.3 to 26.3

Figure C.2: The mean evaluation value of PSO models from the Comparison phase.

For problems of Formulation I in Figure C.2(a), the BCSP model has the highest overall mean evaluation value and all others do not show much difference, which is consistent with the findings of the success rate. Problems 10.53~10.64 were incomplete on $pop = 10$.



(a) Problem set 10



(b) Problems 20.3 to 26.3

Figure C.3: The mean run time of PSO models from the Comparison phase.

The experiments on problems 10.53~10.64 were incomplete with $pop = 10$. The mean run time of all models in Figure C.3(b) grows as the complexity of the problems increases. Particularly, both Discrete models have much higher mean run time. The hikes at problem 10.41 in Figure C.3(a) are discussed in Section 5.4.2. The hike at problem 10.53 comes from *binaryZigzagHop* caused by running out of memory.

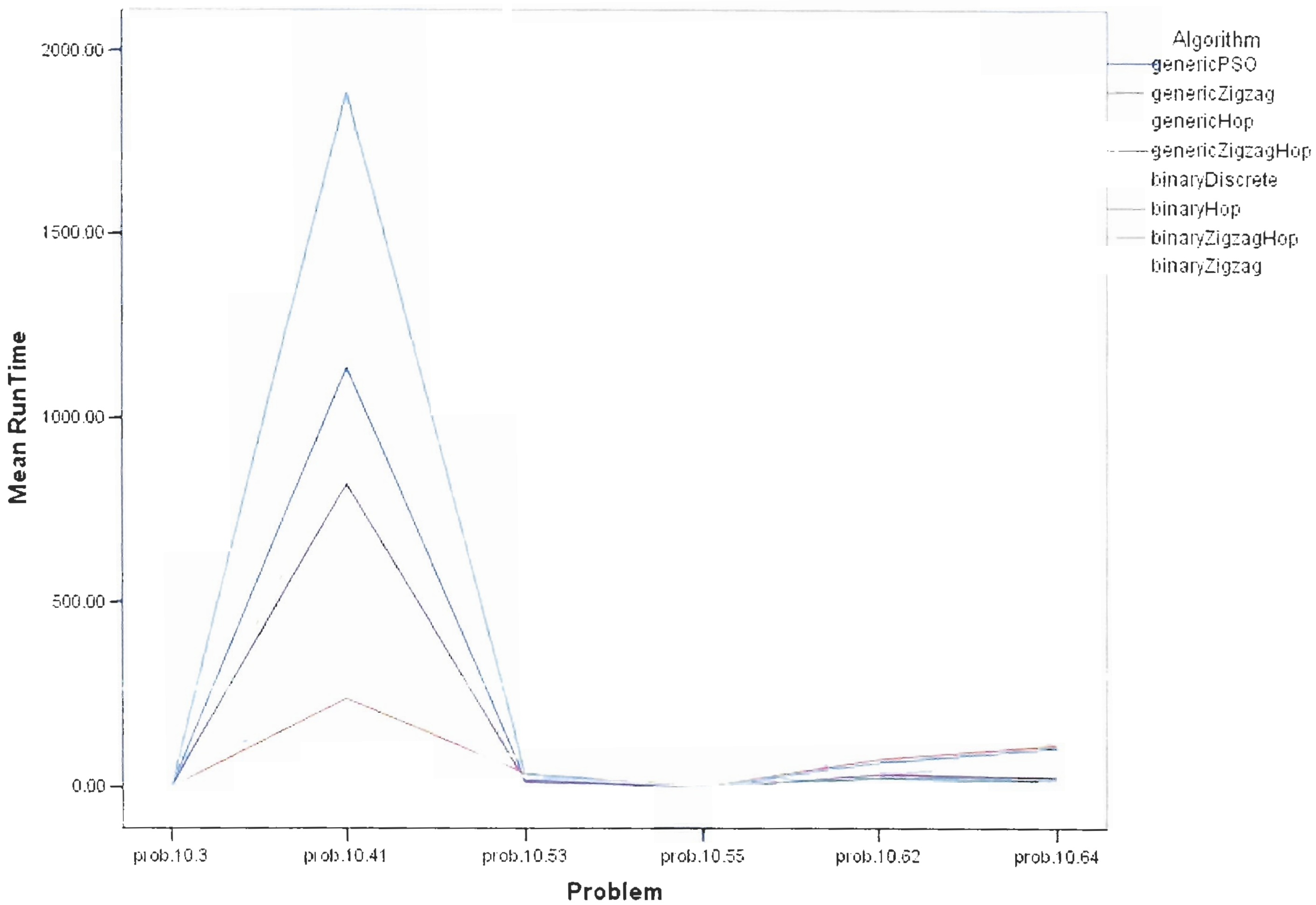


Figure C.4: The mean run time of PSO algorithms using the distance function.

The outcomes are problem set 10 from the Comparison phase. The experiments on problems 10.53~10.64 were incomplete with *pop* = 10. The hikes at problem 10.41 in Figure C.3(a) are discussed in Section 5.4.2.

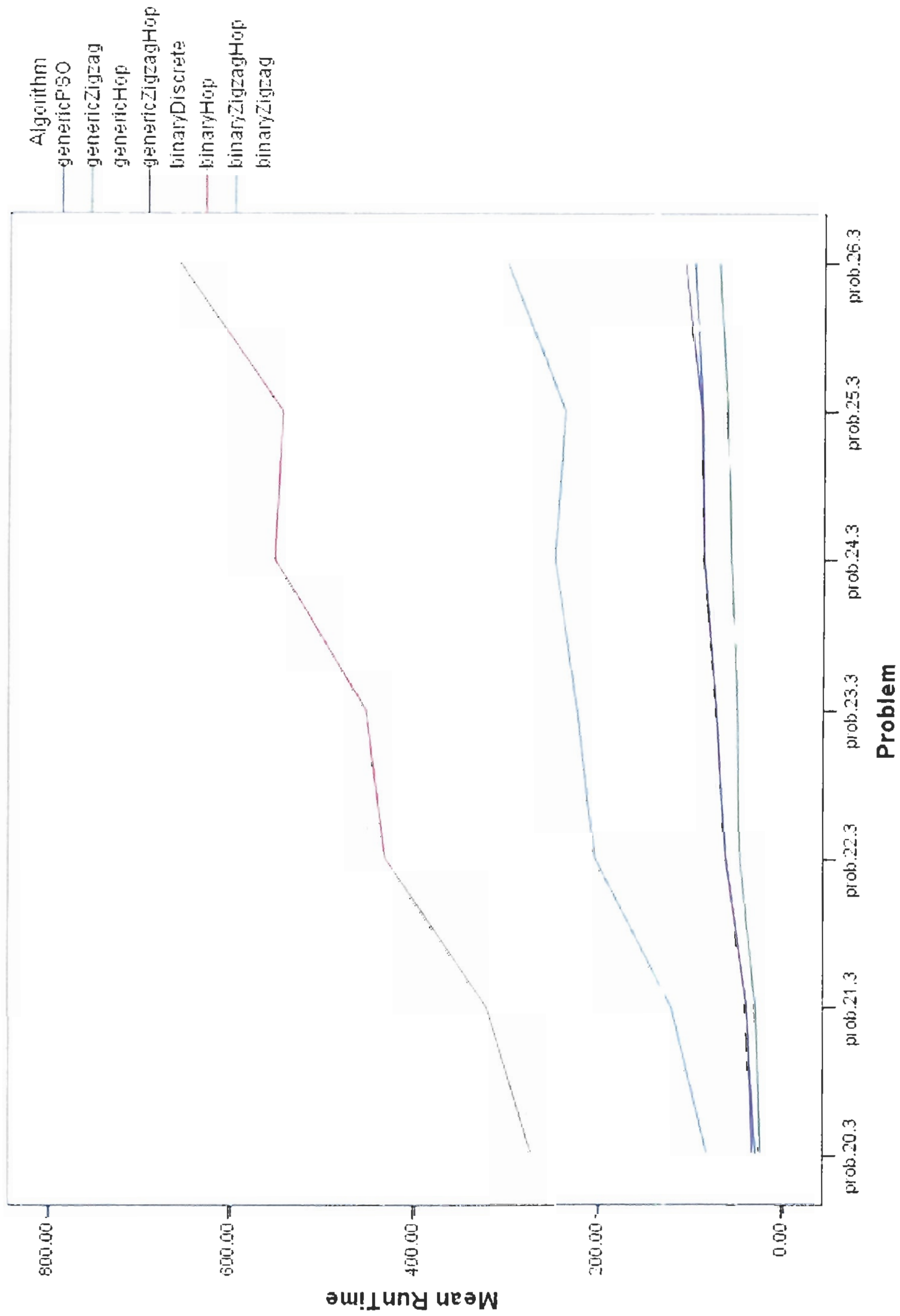


Figure C.5: The mean run time of PSO algorithms using the distance function. The outcomes are problems 20.3~26.3 from the Comparison phase.

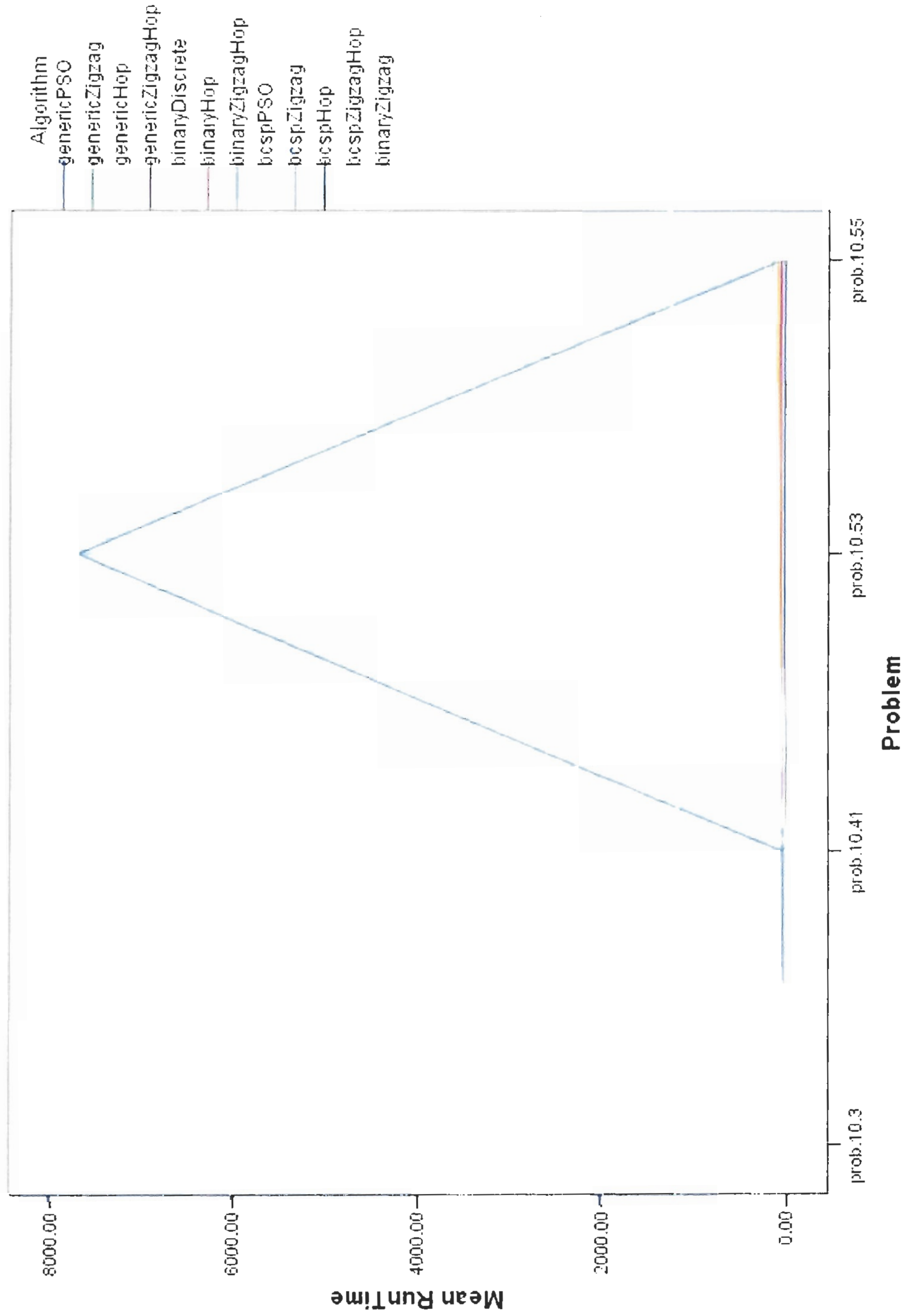


Figure C.6: The mean run time of PSO algorithms using the conflict count function.

The outcomes are problem set 10 from the Comparison phase. The experiments on problems 10.53~10.64 were incomplete with *pop* = 10.

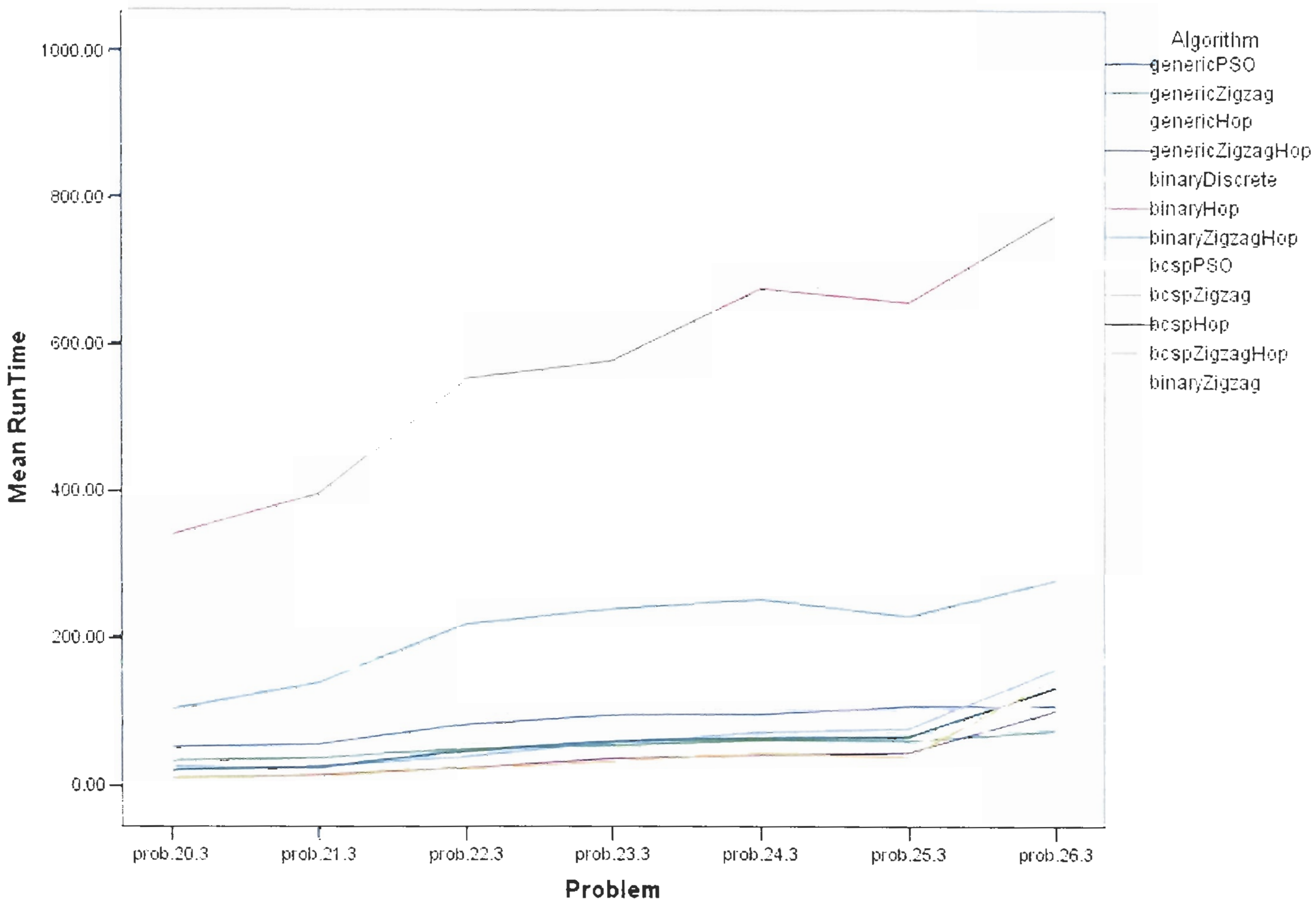
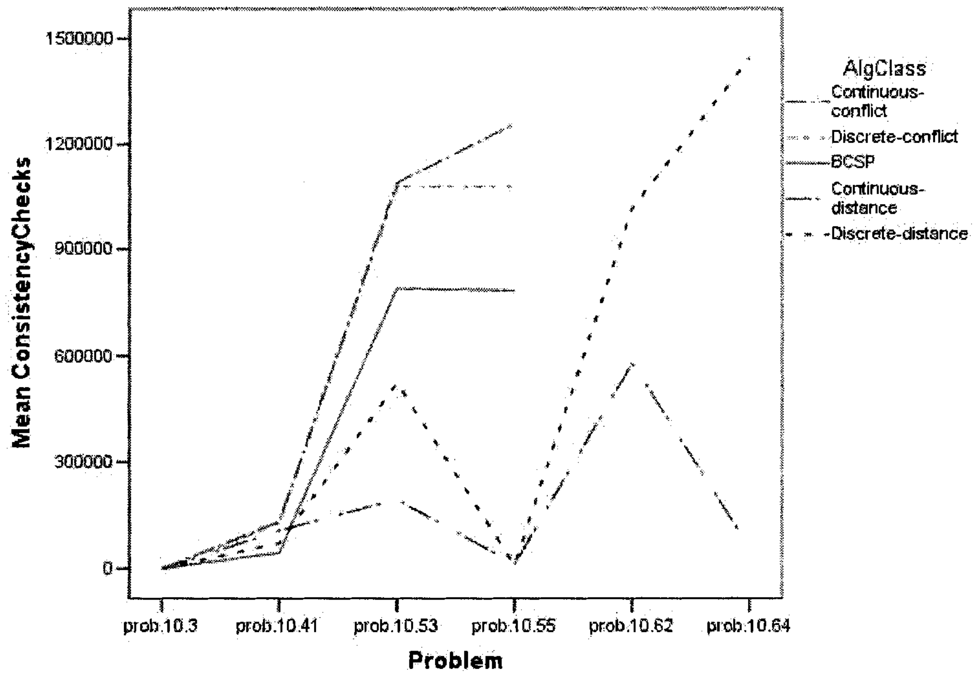
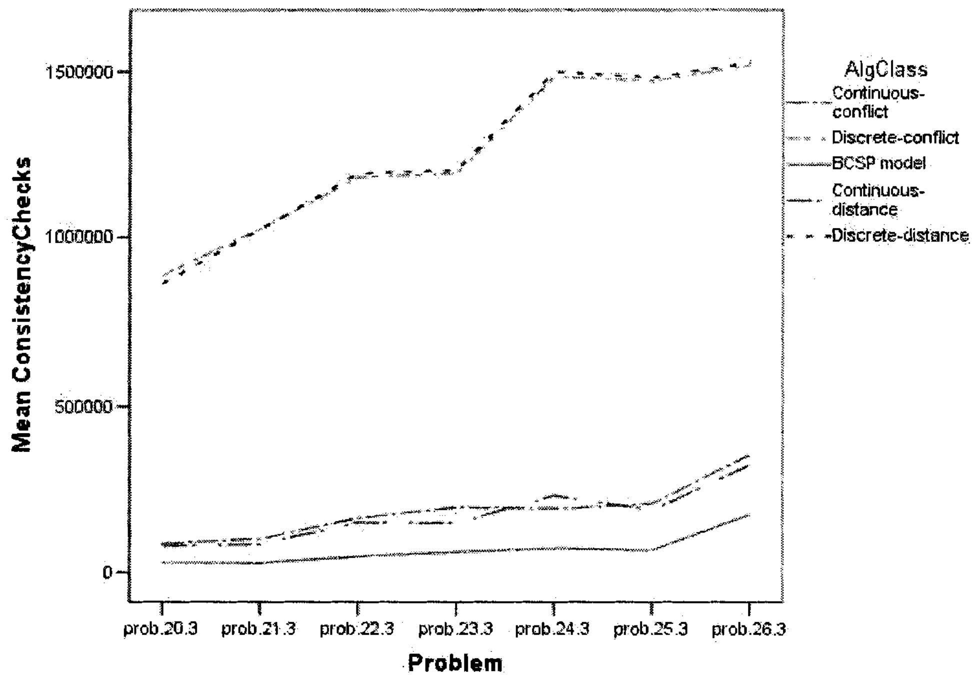


Figure C.7: The mean run time of PSO algorithms using the conflict count function.

The outcomes are problems 20.3∼26.3 from the Comparison phase.



(a) Problem set 10



(b) Problems 20.3 to 26.3

Figure C.8: The mean number of consistency checks of PSO models from the Comparison phase.

The experiments on problems 10.53~10.64 were incomplete with $pop = 10$.

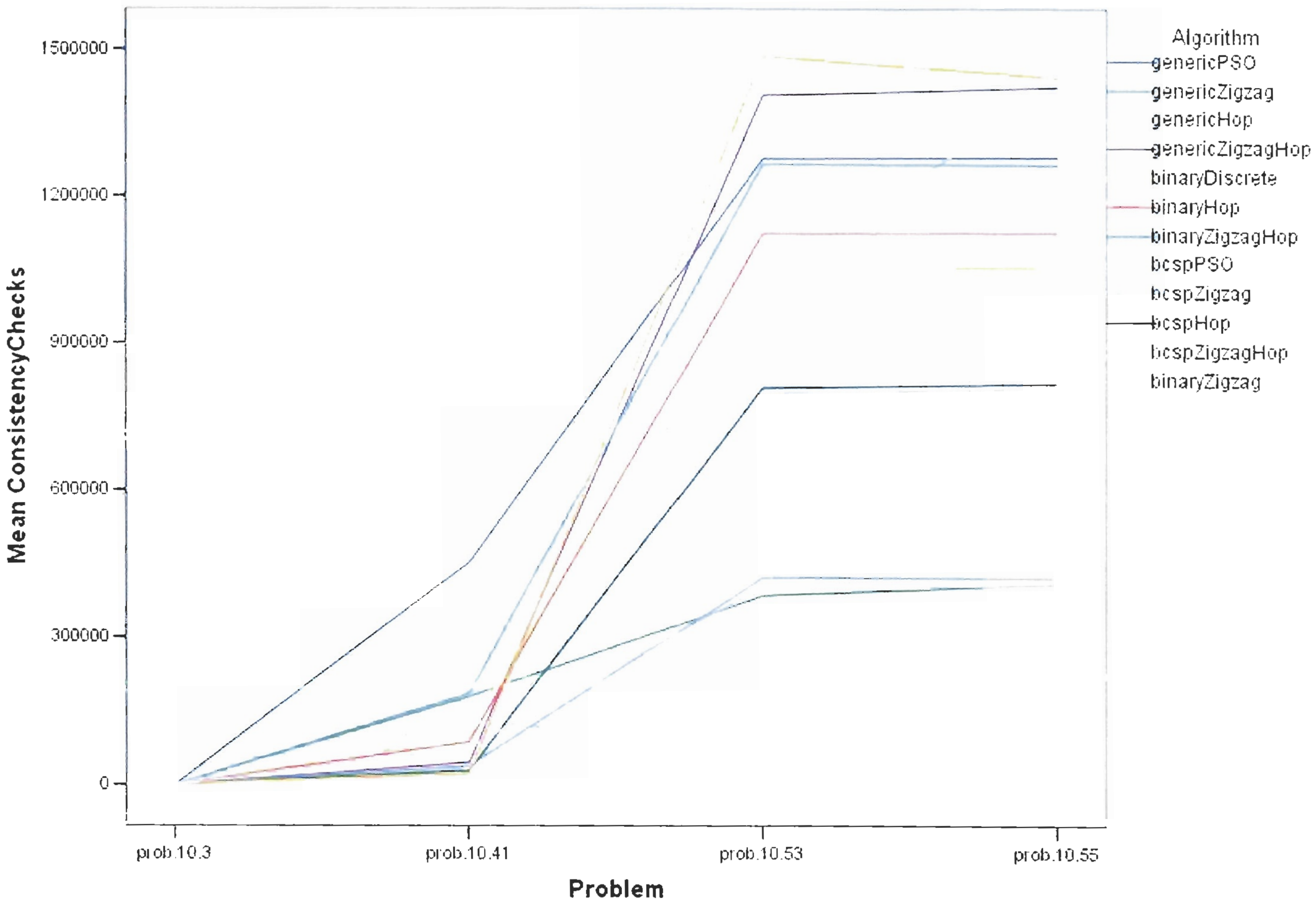


Figure C.9: The mean number of consistency checks of PSO algorithms using the conflict count.

The outcomes are problem set 10 from the Comparison phase. The experiments on problems 10.53~10.64 were incomplete with *pop* = 10.

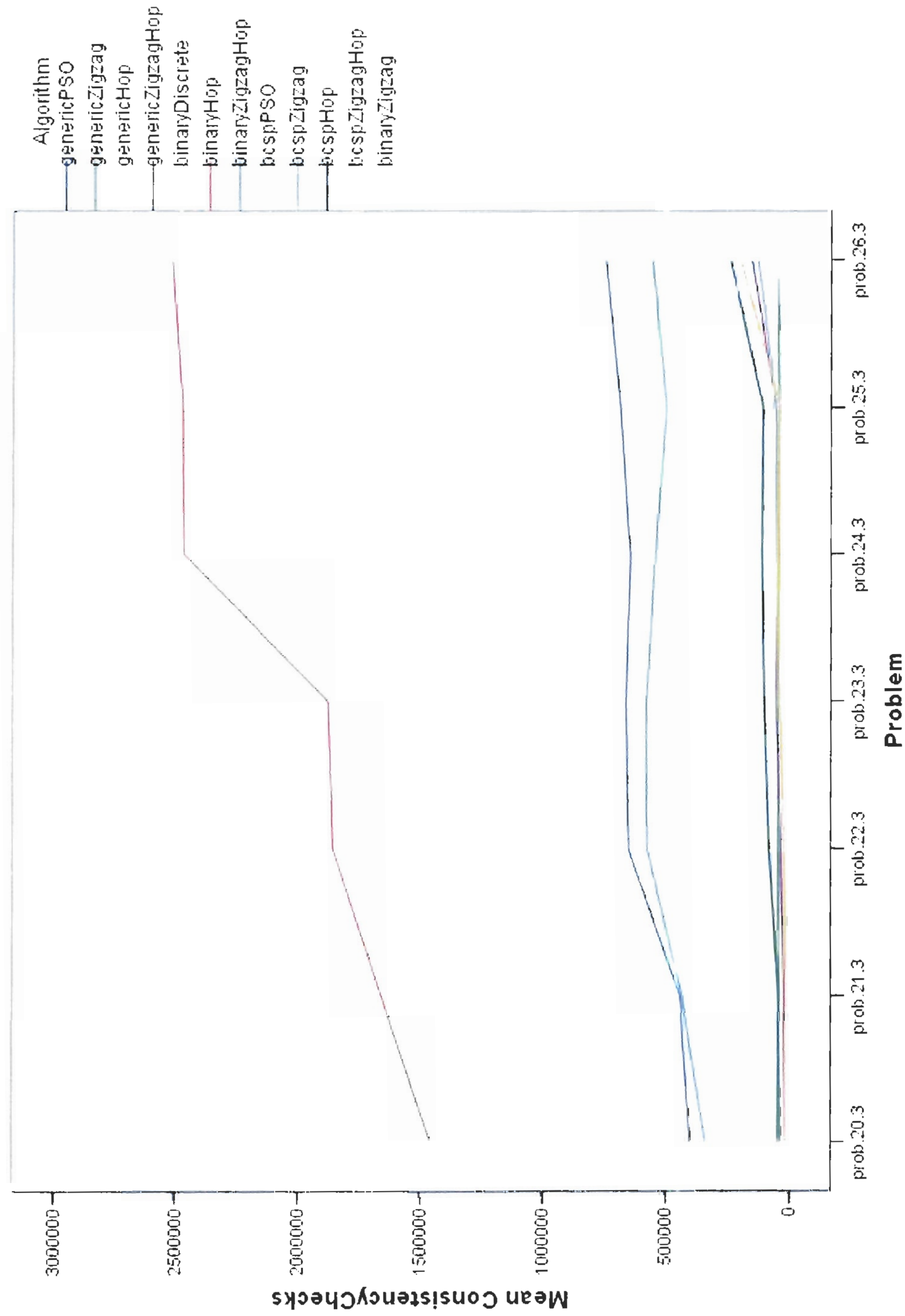


Figure C.10: The mean number of consistency checks of PSO algorithms using the conflict count.

The outcomes are problems 20.3~26.3 from the Comparison phase.

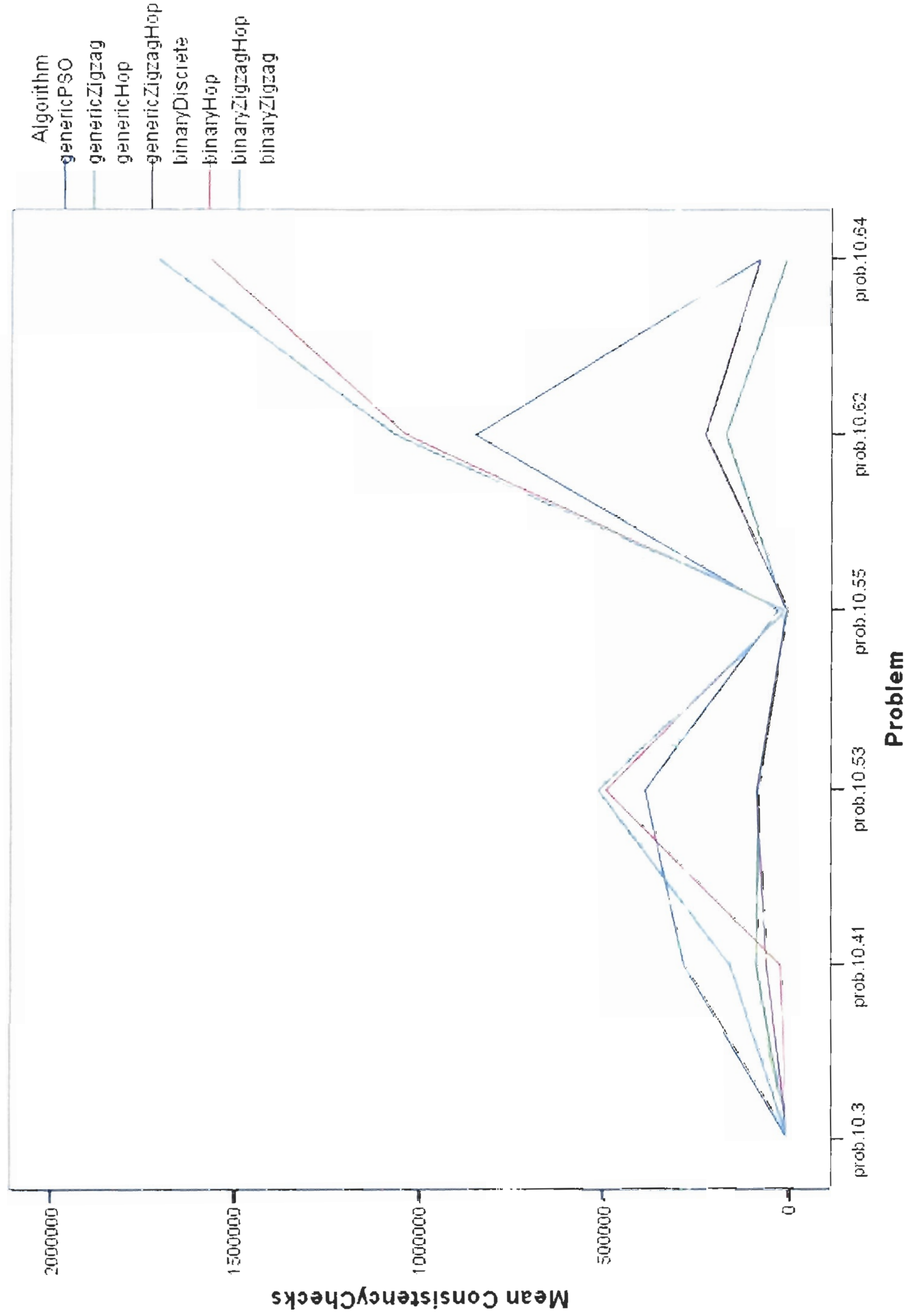


Figure C.11: The mean number of consistency checks of PSO algorithms using the distance function.

The outcomes are problem set 10 from the Comparison phase. The experiments on problems 10.53~10.64 were incomplete with *pop* = 10.

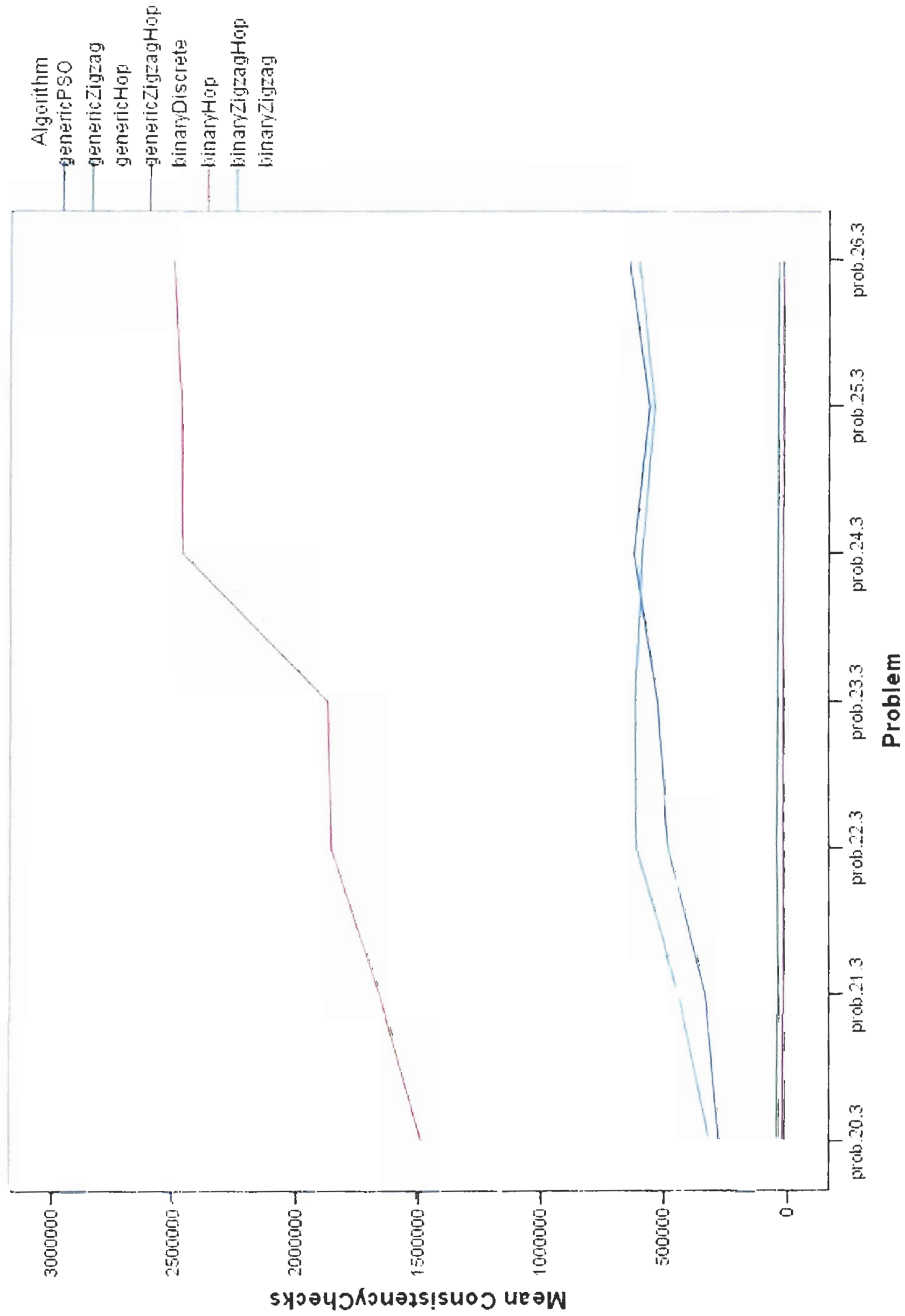
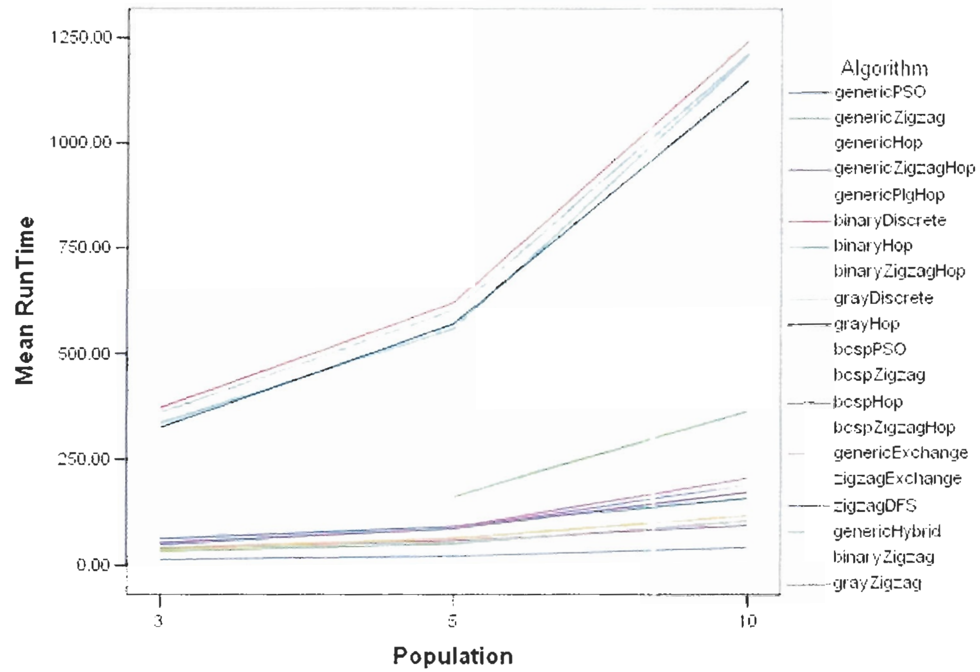
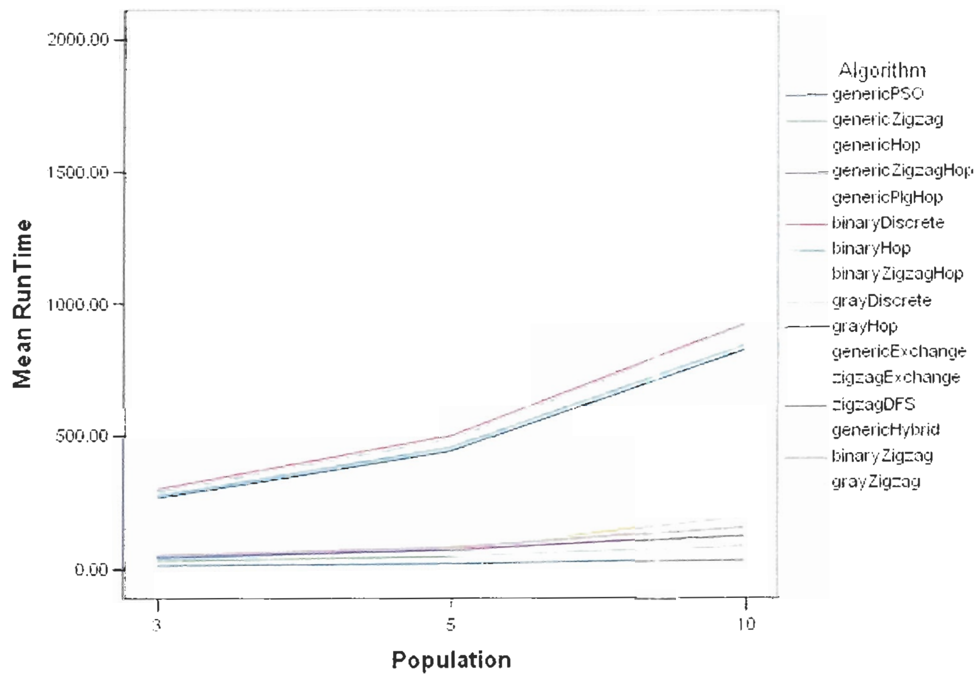


Figure C.12: The mean number of consistency checks of PSO algorithms using the distance function.

The outcomes are Problems 20.3~26.3 from the Comparison phase.

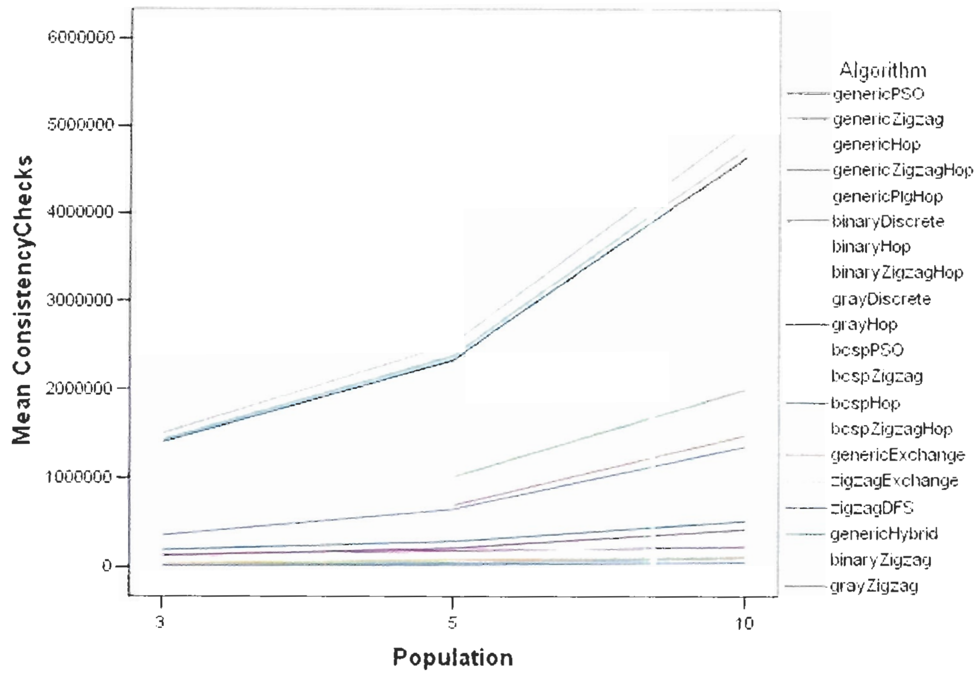


(a) Algorithms using conflict function

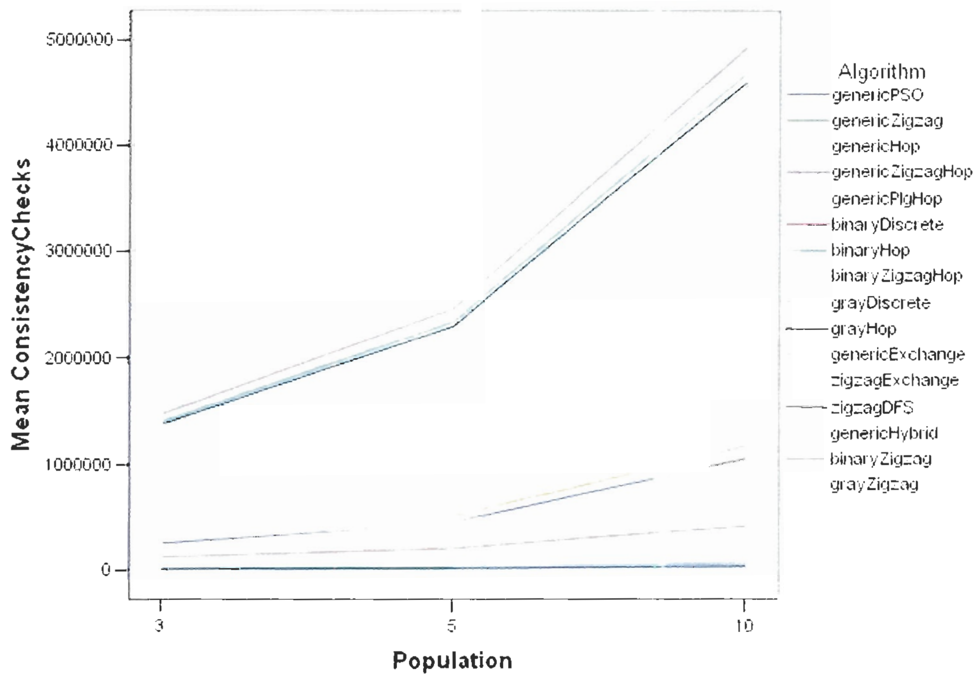


(b) Algorithms using distance function

Figure C.13: The mean run time of PSO populations: problem set 25 from the Comparison phase.



(a) Algorithms using conflict function



(b) Algorithms using distance function

Figure C.14: The mean number of consistency checks of PSO populations: problem set 25 from the Comparison phase.

Bibliography

[1] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinational Optimization and Neural Computing.* John Wiley & Sons, Inc., New York, NY, USA, 1989.

[2] Roman Bartak. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS99)*, volume Part IV, pages 555–564, Prague, June 1999. MatFyzPress.

[3] Gerardo Beni and Jing Wang. Swarm intelligence in cellular robotics systems. In *Proceedings of NATO Advanced Workshop on Robots and Biological System*, June 1989.

[4] Reinaldo A.C. Bianchi and Anna H.R. Costa. Ant-vibra: a swarm intelligence approach to learn task coordination. In G. Bittencourt and G.L. Ramalho, editors, *Proceedings of Advances in Artificial Intelligence: 16th Brazillian Symposium on Artificial Intelligence*, volume 2507 of *Lecture Notes In Computer Science*, pages 195–204, London, UK, November 2002. Springer-Verlag.

[5] Daniel W. Boeringer and Douglas H. Werner. Particle swarm optimization versus genetic algorithms for phased array synthesis. *IEEE Transactions on Antennas and Propagation*, 52:771–779, March 2004.

[6] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: from Natural to Artificial Systems.* Oxford University Press, 198 Madison Avenue, New York, 1999.

[7] R. Brits, AP Engelbrecht, and F. van den Bergh. Solving systems of unconstrained equations using particle swarm optimizatioin. In *Proceedings of the 2002 IEEE International Conference on Systems, Man, and Cybernetics*, Piscataway, NJ, 2002. IEEE Service Center.

[8] Lance D. Chambers. *The Practical Handbook of Genetic Algorithms: Applications.* CRC Press, Boca Raton, FL, USA, second edition, 2000.

[9] Adrian J. Chung. Getting to grips with recursion. *Python Journal*, 2(2), 2001.

[10] Maurice Clerc. The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC'99*, volume 3, pages 1951–1957, Piscataway, NJ, 1999. IEEE Service Center.

[11] Maurice Clerc. Discrete particle swarm optimization: Illustrated by the traveling salesman problem. http://clerc.maurice.free.fr/pso/pso_tsp/Discrete_PSO_TSP.htm, February 2000.

[12] Maurice Clerc and James Kennedy. The particle swarm: Explosion, stability, and convergence in a multi-dimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, Feb 2002.

[13] Genevieve Coath and Saman K. Halgamuge. A comparison of constraint-handling methods for the application of particle swarm optimization to constrained nonlinear optimization problems. In *Proceedings of the 2003 Congress on Evolutionary Computation, CEC'03*, volume 4, pages 2419–2425, Piscataway, NJ, 2003. IEEE Service Center.

[14] Carlos A. Coello Coello. A survey of constraint handling techniques used with evolutionary algorithms. Technical Report Lania-RI-99-04, Laboratorio Nacional de Informática Avanzada, Xalapa, Veracruz, México, 1999.

[15] Carlos A. Coello Coello and Maximino Salazar Lechuga. Mopso: A proposal for multiple objective particle swarm optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation 2002 (CEC2002)*, volume 2, pages 1051–1056, Piscataway, NJ, USA, May 2002. IEEE Press.

[16] B.G.W. Craenen, A.E. Eiben, and E. Marchiori. How to handle constraints with evolutionary algorithms. *Practical Handbook of Genetic Algorithms: Applications*, pages 341–361, 2001.

[17] B.G.W. Craenen, A.E. Eiben, and Jano van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5):424–444, Octobor 2003.

[18] Lawrence Davis. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Pitman/Morgan Kaufmann Publishers Inc., London, 1987.

[19] Lawrence Davis. *Handbook of Genetic Algorithm.* Van Nostrand Reinhold, New York, NY, USA, 1991.

[20] Toby Donaldson, I-Ling Lin, Victor Chen, and Wei Wang. Constraint programming in python. presented at VanPy Workshop 2004, 2004.

[21] Marco Dorigo. About ant colony optimization, December 2004.
http://iridia.ulb.ac.be/~mdorigo/ACO/about.html.

[22] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: an autocatalytic optimizing process. Technical Report No. 91–016 Revised, Department of Electronic Information, Politecnico di Milano, Politecnico di Milano, Italy, 1991.

[23] Russell Eberhart and Yuhui Shi. Particle swarm optimization: Developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation, CEC2001*, volume 1, pages 81–86, Piscataway, NJ, 2001. IEEE Service Center.

[24] Russell C. Eberhart and Yuhui Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation, CEC2000*, volume 1, pages 84–88, Piscataway, NJ, 2000. IEEE Service Center.

[25] Andries P. Engelbrecht. *Computational Intelligence: an Introduction.* John Wiley & Sons, Ltd, West Sussex, England, 2002.

[26] Kent Fitch. Particle swarm optimization (pso) visualization, April 2004. http://www.projectcomputing.com/resources/psovis/.

[27] Alan M. Frisch, Ian Miguel, and Toby Walsh. CGRASS: A system for transforming constraint satisfaction problem. In *Recent Advances in Constraints: Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, volume LNAI 2627 of *Lecture Notes in Computer Science*, pages 15–30, New York, NY, USA, 2003. Springer-Verlag Berlin Heidelberg.

[28] Yoshikazu Fukuyama. Fundamentals of particle swarm optimization techniques. In *IEEE PES Tutorial on Modern Heuristic Optimization Techniques with Application to Power Systems*, chapter 5. IEEE Service Center, Piscataway, NJ, 2001.

[29] Zwe-Lee Gaing. Particle swarm optimization to solving the economic dispatch considering the generator constraints. *IEEE Transactions on Power Systems*, 18(3):1187–1195, August 2004.

[30] John Gary Gaschnig. A general backtrack algorithm that eliminates most redundant test. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, volume 1, page 457, Cambridge, MA, USA, 1977. Morgan Kaufmann.

[31] John Gary Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie Mello University, Pittsburg USA, May 1979.

[32] Ian Gent and Toby Walsh. Computational phase transitions from real problems. In *Proceedings of the 8th International Symposium on AI*, pages 356–364, 1995.

[33] S.P. Ghoshal. Optimizations of pid gains by particle swarm optimizations in fuzzy based automatic generation control. *Electric Power Systems Research*, 72:203–212, December 2004.

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

[35] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition, January 1989.

[36] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, October 1965.

[37] Robter M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[38] Pascal Van Hentenryck and Vijay Saraswat. Strategic directions in constraint programming. *ACM Computing Surveys (CSUR)*, 28(4):701–726, 1996.

[39] F. Heppner and U. Grenander. A stochastic nonlinear model for coordinated bird flocks. In S. Krasner, editor, *The Ubiquity of Chaos*. AAAS Publications, Washington, DC, 1990.

[40] John H. Holland. *Adaptation in Natural and Artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, Ann Arbor, June 1975.

[41] Xiaohui Hu. Particle swarm optimization, December 2004. http://www.swarmintelligence.org/index.php.

[42] Xiaohui Hu and Russell Eberthart. Solving constrained nonlinear optimization problems with particle swarm optimization. In *Proceedings of the Sixth World Multiconference on Systemics, Cybernetics and Informatics (SCI2002)*, volume V, 2002.

[43] Xiaohui Hu, Russell C. Eberthart, and Yuhui Shi. Swarm intelligence for permutation optimization: a case study on n-queens problem. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003 (SIS2003)*, pages 243–246. IEEE, April 2003.

[44] Xiaohui Hu, Yuhui Shi, and Russell Eberhart. Recent advances in particle swarm. In *Proceedings of the 2004 Congress on Evolutionary Computation CEC2004*, volume 1, pages 90–97, Piscataway, NJ, June 2004. IEEE Service Center.

[45] ILOG Inc. ILOG solver, August 2004. http://www.ilog.com/products/solver/.

[46] Peter G. Jeavons, D.A. Cohen, and M. Cooper. Constraints, consistency and closure. *Artificial Intelligence*, 101(1-2):251–265, March 1998.

[47] Peter G. Jeavons, Nick W. Dunkin, and Joe E. Bater. Why higher order constraints are necessary to model frequency assignment problem. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence, ECAI98*. John Wiley & Sons, Ltd., 1998.

[48] J. Jerald, P. Asokan, G. Prabaharan, and R. Saravanan. Scheduling optimisation of flexible manufacturing systems using particle swarm optimisation algorithm. *The International Journal of Advanced Manufacturing Technology*, pages 399–408, 2004.

[49] Ulrich Junker. Preference programming for configuration. *Workshop on Configuration, IJCAI-01*, pages 50–56, August 2001.

[50] James Kennedy. The behavior of particles. In V. William Porto, N. Saravanan, Donald E. Waagen, and A. E. Eiben, editors, *Proceedings of the 1998 Evolutionary Programming Conference*, volume 1447 of *Lecture Notes In Computer Science*, pages 581–589, London, UK, March 1998. Springer-Verlag.

[51] James Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC99*, volume 3, pages 1931–1938, Piscataway, NJ, 1999. IEEE Service Center.

[52] James Kennedy. *Recent Developments in Biologically Inspired Computing*, chapter 10 Particle Swarms: Optimization Based on Sociocognition. Idea Group Publishing, 701 E Chocolate Avenue, Suite 200, Hershey PA17033, 2005.

[53] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks (ICNN'95)*, volume 4, pages 1942–1948, Piscataway, NJ, USA, 1995. IEEE Service Center.

[54] James Kennedy and Russell C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Proceedings of the Conference on Systems, Man, and Cybernetics*, pages 4104–4109, 1997.

[55] James Kennedy, Russell C. Eberhart, and Yuhui Shi. *Swarm Intelligence : Collective, Adaptive...* Morgan Kaufman, San Francisco, second edition, 2001.

[56] Po-Chang Ko and Ping-Chen Lin. A hybrid swarm intelligence based mechanism for earning forecast. In *Proceedings of the 2nd International Conference on Information Technology for Application (ICITA2004)*, pages 193–198. Macquarie Scientific Publishing, 2004.

[57] Vipin Kumar. Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, Spring 1992.

[58] The Artificial Intelligence Laboratory. Jcl - the java constraints library, March 2004. http://liawww.epfl.ch/JCL/index.htm.

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

[60] Elena Marchiori and Adri Steenbeak. A genetic local search algorithm for random binary constraint satisfaction problems. In *Proceedings of the 14th ACM symposium on Applied Computing (SAC 2000)*, pages 458–462, New York, NY, 2000. ACM Press.

[61] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, Cambridge, Massachusetts, 1998.

[62] Alex Martelli and David Ascher. *Python Cookbook.* O'Reilly, Beijing, Farnham, 2002.

[63] Zbigniew Michalewicz, Dipankar Dasgupta, Rodolphe G. Le Riche, and Marc Schoenauer. Evolutionary algorithms for constrained engineering problems. *Computers & Industrial Engineering Journal*, 30(4):851–870, September 1996.

[64] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90)*, volume 2, pages 17–24. AAAI Press/The MIT Press, 1990.

[65] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence: Special Volume on Constraint Based Reasoning*, 58(1-3):161–205, December 1992.

[66] Musi-Cal. Python performance tips, 2004. http://musi-cal.mojam.com/~skip/python/fastpython.html.

[67] Shigenori Naka, Takamu Genji, Toshiki Yura, and Yoshikazu Fukuyama. Practical distribution state estimation using hybrid particle swarm optimization. In *Proceedings of IEEE Power Engineering Society Winter Meeting, 2001*, volume 2, pages 815–820, Piscataway, NJ, 2001. IEEE Service Center.

[68] K. Nara and Y. Mishima. Particle swarm optimization for fault state power supply reliability enhancement. In *Proceedings of IEEE International Conference on Intelligent Systems Application to Power Systems, ISAP2001*, pages 172–176. IEEE, June 2001.

[69] Travis Oliphant. Numerical python, 2004. http://numeric.scipy.org/.

[70] R.H.J.M. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm.* SECS72. Kluwer Academic Publishers, Boston, June 1989.

[71] Ulrich Paquet and Andries P. Engelrecht. A new particle swarm optimizer for linearly constrained optimization. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of IEEE Congress on Evolutionary Computation 2003 (CEC 2003)*, pages 227–233, Piscataway, NJ, USA, 2003. IEEE Press.

[72] Konstantinos E. Parsopoulos and Michael N. Vrahatis. Particle swarm optimization method for constrained optimization problems. *Intelligent Technologies - Theory and Applications: New Trends in Intelligent Technologies*, 76:214–220, 2002.

[73] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence: an international journal*, 9(3):268–299, August 1993.

[74] Patrick Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report 95–177, University of Strathclyde, Department of Computer Science, University of Strathclyde, UK, 1995.

[75] Jean-Francois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In *Proceedings of the 1995 International Logic Programming Symposium (ILPS'95)*, Logic Programming, Research Reports and Notes, pages 513–527. The MIT Press, January 1996.

[76] A. Ratnaweera, H. Watson, and S.K. Halgamuge. Optimization of value timing events of internal combustion engines with particle swarm optimization. In *Proceedings of the 1th International Conference on Fuzzy Systems and Knowledge Discovery 2002, FSKD2002*, pages 264–268, 2003.

[77] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[78] Robert G. Reynolds, Zbigniew Michalewicz, and M. Cavaretta. Using cultural algorithms for constraint handling in genocop. In D. Fogel, J. McDonnell, and R. Reynolds, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, Complex Adaptive Systems, pages 289–305. MIT Press, 1995.

[79] Armin Rigo. Psyco project, 2004. http://psyco.sourceforge.net/.

[80] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In Luigia Carlucci Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence, ECAI'90*, pages 550–556, Stockholm, 1990. Pitman.

[81] Pierre Roy, Anne Liret, and Francois Pachet. The framework approach for constraint satisfaction. *ACM Computing Surveys (CSUR)*, 32(1es):13, March 2000.

[82] Pierre Roy and Francois Pachet. Reifying constraint satisfaction in smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 10(4):43–51, 63, July/August 1997.

[83] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, second edition, 2003.

[84] Daniel Sabin and Rainer Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems*, 13(4):42–49, July/August 1998.

[85] J. Salerno. Using the particle swarm optimization technique to train a recurrent neural model. In *Proceedings of the 9th International Conference on Tools with Artificial Intelligence, ICTAI'97*, pages 45–49, 1997.

[86] Miguel A. Salido and Federico Barber. Disjunction of non-binary and numeric constraint satisfaction problems. In *Proceedings of the 5th Catalonian Conference on Artificial Intelligence, CCIA 2002*, volume LNAI 2504, pages 159–172, New York, NY, October 2002. Springer-Verlag Berlin Heidelberg.

[87] Ayed Salman, Imtiaz Ahmad, and Sabah Al-Madani. Particle swarm optimization for task assignment problem. *Journal of Microprocessors and Microsystems*, 26:363–371, January 2002.

[88] Luk Schoofs and Bart Naudts. Ant colonies are good at solving constraint satisfaction problems. In *Proceedings of the 2000 Congress on Evolutionary Computation, CEC2000*, volume 2, pages 1190–1195, Piscataway, NJ, 2000. IEEE Service Center.

[89] Luk Schoofs and Bart Naudts. Solving csps with ant colonies. In Hans-Pau Schwefel, Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, and Juan Julian Merelo, editors, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, pages 16–20, Paris, France, 2000. Springer Verlag.

[90] Luk Schoofs and Bart Naudts. Swarm intelligence on the binary constraint satisfaction problem. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)*, volume 2, pages 1444–1449. IEEE, May 2002.

[91] Barry R. Secrest and Gary B. Lamont. Visualizing particle swarm optimization - gaussian particle swarm optimization. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium, 2003. SIS'03*, pages 198–204, Piscataway, NJ, USA, April 2003. IEEE Service Center.

[92] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 1, pages 290–295, San Francisco, USA, 1993. Morgan Kaufmann Publishers.

[93] Yuhui Shi. A modified particle swarm optimizer. In *Proceedings of IEEE Congress on Evolutionary Computation, CEC1998*, pages 69–73, Piscataway, NJ, 1998. IEEE.

[94] Yuhui Shi and Russell Eberhart. Fuzzy adaptive particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation 2001*, volume 1, pages 101–106, Piscataway, NJ, 2001. IEEE Service Center.

[95] Yuhui Shi and Russell C. Eberhart. Parameter selection in particle swarm optimization. In V. William Porto, N. Saravanan, Donald E. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII, the Proceedings of the 7th International Conference*

on Evolutionary Programming, EP98, volume 1447 of *Lecture Notes In Computer Science*, pages 591–600, London, UK, March 1998. Springer-Verlag.

[96] Yuhui Shi and Russell C. Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation, CEC99*, volume 3, pages 1945–1950, Piscataway, NJ, 1999. IEEE Service Center.

[97] Barbara M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In A.G.Cohn, editor, *Proceedings of ECAI'94*, pages 100–104. John Wiley and Sons, 1994.

[98] Barbara M. Smith. A tutorial on constraint programming. Technical Report 95.14, University of Leeds, School of Computer Studies, University of Leeds, April 1995.

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

[100] Tiago Sousa, Arlindo Silva, and Ana Neves. Particle swarm based data mining algorithms for classification tasks. *Parallel Computing*, 30:767–783, May 2004.

[101] Olaf Steinmann, Antje Strohmaier, and Thomas Stutzle. Tabu search vs. random walk. In Christopher Habel Gerhard Brewka and Bernhard Nebel, editors, *Advances in Artificial Intelligence: Proceedings of 21st Annual German Conference on Artificial Intelligence, in KI-97*, volume LNCS1303, pages 337–348. Springer Verlag, 1997.

[102] Kostas Stergiou. *Particle Swarms - Extensions For Improved Local, Multi-Modal, and Dynamic Search in Numerical Optimization.* PhD thesis, Department of Computer Science, the University of Strathclyde, Glasgow, Scotland, January 2001.

[103] Markus Stumptner, Michel Aldanondo, Gerhard Friedrich, Esther Gelle, Timo Soininen, and Reijo Sulonen. Preface. In *Configuration, the Proceedings of the Workshop at ECAI 2000, the 14th European Conference on Artificial Intelligence*, page vii, August 2000.

[104] Vincent Tam and K.T. Ma. Applying genetic algorithms and other heuristic methods to handle pc configuration problems. In V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - Proceedings of the International Conference on Computational Science, ICCS 2001*, volume 2074 of *Lecture Notes in Computer Science*, pages 439–446, New York, NY, USA, May 2001. Springer-Verlag Berlin Heidelberg.

[105] Vincent Tam and K.T. Ma. Optimizing personal computer configurations with heuristic-based search methods. *Artificial Intelligence Review*, 17(2):129–140, April 2002.

[106] Vipul Tandon. Closing the gap between cad/cam and optimized cnc end milling. Master's thesis, Purdue School of Engineering and Technology, Indiana University Purdue University, Indianapolis, 2001.

[107] Peter Tarasewich and Patrick R. McMullen. Swarm intelligence: Power in numbers. *Communications of the ACM*, 45(8):62–67, 2002.

[108] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press Inc., San Diego, California, USA, 1993.

[109] Edward P.K. Tsang and Terry Warwick. Applying genetic algorithms to constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI 1990)*, pages 649–654. Pitman Publishing, 1990.

[110] Supiya Ujjin and Peter J. Bentley. Particle swarm optimization recommender system. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003 (SIS'03)*, pages 124–131, Piscataway, NJ, 2003. IEEE Service Center.

[111] F. van den Bergh and A.P. Engelbrecht. A new locally convergent particle swarm optimizer. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics SMC 2002*, pages 96–101, Piscataway, NJ, 2002. IEEE Service Center.

[112] Jano van Hemert. Constraint satisfaction problems and evolutionary computation: A reality check. In A. ven den Bosch and H. Weigand, editors, *Proceedings of the 12th Belgium/Netherlands Conference on Artificial Intelligence, BNAIC'00*, pages 267–274, De Efteling, Tilburg, The Netherlands, November 2000. Dutch and the Belgian AI Association.

[113] Jano van Hemert and Christine Solnon. A study into ant colony optimization, evolutionary computation and constraint programming on binary constraint satisfaction problems. In Jens Gottlieb and Günther R. Raidl, editors, *Proceedings of the 4th European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP2004*, volume 3004 of *Lecture Notes in Computer Science*, pages 114–123, New York, NY, USA, 2004. Springer-Verlag Heidelberg.

[114] Guido van Rossum. Python performance tips, 2005. http://wiki.python.org/moin/PythonSpeed/PerformanceTips.

[115] Jakob Vesterstrøm and Jacques Riget. Particle swarms – extensions for improved local, multi-modal, and dynamic search in numerical optimization. Master's thesis, Department of Computer Science, University of Aarhus, May 2002.

[116] Hirotaka Yoshida, Kenichi Kawata, Yoshikazu Fukuyama, Shinichi Takayama, and Yosuke Nakanishi. A particle swarm optimization for reactive power and voltage control considering voltage security assessment. *IEEE Transactions on Power Systems*, 15(4):1232–1239, November 2000.

[117] Yangyang Zhang, Chunlin Ji, Ping Yuan, Manlin Li, Chaojin Wang, and Guangxing Wang. Particle swarm optimization for base station placement in mobile communication. In *Proceedings of 2004 IEEE International Conference on Networking, Sensing and Control*, volume 1, pages 428–432, Piscataway, NJ, 2004. IEEE Service Center.

[118] Bo Zhao and Yi jia Cao. Multiple objective particle swarm optimization technique for economic load dispatch. *Journal of Zhejiang University Science, JZUS*, 6A(5):420–427, May 2005.

[119] Ying Zhao and Junli Zheng. Particle swarm optimization algorithm in signal detection and blind extraction. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN'04*, pages 37–41. IEEE, 2004.