University of Strathclyde
Department of Computer and Information Sciences

Extracting and Exploiting Interaction Information in Constraint-Based Local Search

by
Alastair Neil Andrew

A thesis presented in fulfilment of the requirements for the degree of

Doctor of Philosophy

2014

To my Grandparents, Peter Neil Leitch and Christina Andrew, who supported me from the outset but sadly never got to witness the conclusion.

Acknowledgements

I could not have produced this thesis without the support and encouragement of numerous people. Firstly, my supervisor, Dr John Levine, whose cheery disposition and sage advice were always appreciated. Even when I felt lost, he seemed to have faith that I knew what I was doing (or at least pretended to).

Professors Derek Long and Maria Fox for setting up the Strathclyde Planning Group and creating an inspiring environment to study in. They showed me the level of hard-work and commitment that it takes to become a world-class researcher. Their presence attracted many foreign visitors and gave me the opportunity to meet some of the community's big figures. The team they assembled was also excellent; I'd like to thank all the members of the Strathclyde Planning Group. In particular, Dr Peter Gregory who manages to be both an endlessly distracting yet infuriatingly interesting individual. The Doctors Coles, Andrew and Amanda, who consistently produced a stream of great work and yet still had time to graciously answer my beginner's questions about Unix and LaTeX. Dr Tommy Thompson, a fellow Renfrewshire émigré, whose work ethic and good humour were always appreciated.

I'd like to thank all the PhD students who came before me (Dr Mohammed Newton, Dr Johnson Stewart, Dr Ric Glassey) and those following after me (Alan Lindsay, David Bell, David Pattison, Chris Foley, and Martin Goodfellow). Their friendship has helped immeasurably over the years.

I'd like to thank all the staff in the Department of Computer and Information Sciences who made Strathclyde such a welcoming place to study. Their enthusiasm for computing and their various subjects rubbed off on me and I hope I've been able to add to that too. Not just the teaching staff but all the others who get overlooked yet play a critical role in the functioning of the department. Especially the

System Support team (Ian Gordon, Kenny Forte, and Duncan Sinclair) who endured—with good grace—my regular hassling to update the labs' software.

Out with the department I'd like to thank the staff at the University of Strathclyde Sport and Recreation Centre. Particularly Anne-Marie Hughes for her no-holds-barred approach to exercise, and Marcus Dowds for running the weekly circuits session. Thanks also to the Monday Running Club. Dr Murray Wood, Professor Andy Marshall, Dr Geoff McKay, Kenny Forte, Alan Lindsay (and guest runners) provided a friendly mixture of athletic competition (and good-natured abuse) that helped me improve my personal bests in all distances from the 5 km to the marathon. Hopefully through their example I've managed to stave off some of the worse effects of what could be a sedentary occupation.

I've really appreciated the feedback and encouragement I've received from all the members of the Planning and Constraints communities that I've met at conferences and workshops. Special thanks to Pascal Van Hentenryck and Laurent Michel for creating the COMET language and helping answer my queries whilst I was trying to master it.

Thanks to all my friends and everyone who was interested in how I was getting on but didn't ask! I'm particularly grateful to all the people who have taken time to read drafts and give me their feedback.

My parents, Jean & David Andrew, without whom I couldn't have completed this (nor felt bold enough to begin). I'm sure I've caused them no end of consternation, and they will probably never understand why I chose to do it, but from their example I learned the importance of doing a job properly, regardless of the size, and taking a pride in your work. I find there's a sense of achievement that you can only get by pushing past the limits of what you thought you were capable of. As well as my Mum and Dad, I'm grateful to my whole family: my sister Karen; my Gran, Margaret; and my Aunt Fiona and Uncle Gordon.

And lastly I'd like to thank my partner, Rachel, and our cat Gunner. They never complained about the late nights or all the times when I was lost in my thoughts; for that I am eternally grateful.

Contents

List of Figures

List of Tables

Abstract

Local Search is a simple and effective approach for solving complex constrained combinatorial problems. To maximise performance, Local Search can utilise problem-specific information and be hybridised with other algorithms in an often intricate fashion. This results in algorithms that are tightly coupled to a single problem and difficult to characterise; experience gained whilst solving one problem may not be applicable in another. Even if it is, the translation can be a non-trivial task offering little opportunity for code reuse. Constraint Programming (CP) and Linear Programming (LP) can be applied to many of the same combinatorial problems as Local Search but do not exhibit these restrictions. They use a different paradigm; one where a problem is captured as a general model and then solved by a independent solver. Improvements to the underlying solver can be harnessed by any model. The CP community show signs of moving Local Search in this direction; Constraint-Based Local Search (CBLS) strives to achieve the CP ideal of "*Model + Search*". CBLS provides access to the performance benefits of Local Search without paying the price of being specific to a single problem.

This thesis explores whether information to improve the performance of CBLS can be automatically extracted and exploited without compromising the independence of the search and model. To achieve these goals, we have created a framework built upon the CBLS language COMET. This framework primarily focusses on the interface between two core components: the constraint model, and the search neighbourhoods. Neighbourhoods define the behaviour of a Local Search and how it can traverse the search space. By separating the neighbourhoods from the model, we are able to create an independent analysis component. The first aspect of our work is to uncover information about the interactions between the constraint model and the search neighbourhoods. The second goal is to look at how information about the behaviour of neighbourhoods—with respect to a set of constraints—can be used within the search process. In particular, we concentrate on enhancing a form of Local Search called Variable Neighbourhood Search (VNS) allowing it to make dynamic decisions based upon the current search state. The resulting system retains the domain independence of model-based solution technologies whilst being able to configure itself automatically to a given problem. This reduces the level of expertise required to adopt CBLS and provides users with another potential tool for tackling their constraint problems.

A visual representation of this thesis' contents generated by Wordle™ at `http://www.wordle.net`.

Chapter 1

Introduction

Work as if you live in the early days of a better nation.

Alasdair Gray
Writer & Artist

The Cubist artist Pablo Picasso once dismissed computers as useless because "*they can only give you answers*". A noble sentiment, and certainly a pithy sound-bite, but Picasso seems to do answers a disservice. The natural inquisitiveness of humans means the world is unlikely to be left in want of questions; answers, however, are a rarer and more valuable commodity. Computers can answer problems that are impractical, or even impossible, to calculate manually.

The term Artificial Intelligence (AI) has gripped the public's imagination since the field's mainstream inception in the early 1960s through the media's endless depictions of a future filled with human-like robots and despotic computers. The failure of ambitious early projects, like automatic machine translation or natural language generation, to live up these overhyped ideals has created the perception that AI remains more science-fiction than actual science. The reality is that whilst prestige projects, like domestic robots, remain a way off, the less glamorous—but ultimately more practical—applications are all around us. From the facial recognition in modern digital cameras to the shift rosters of hospital staff, AI can discreetly appear in them all.

Of the issues currently facing the world, the control of energy consumption—and its related environmental consequences—seem the most pressing. Making effective use of resources—be that of energy, capital, time, or labour—will always be of practical economic interest. These resource assignment problems appear in many forms such as: staff rostering, production scheduling, timetabling, delivery routing, etc. When resources are plentiful, solving these problems is relatively easy; as resources become scarcer finding solutions that respect these tight restrictions becomes far more challenging. The problems that we are interested in have two defining characteristics: firstly, they are combinatorial in nature (i.e. there are many potential configurations / decisions); secondly, there exists a set of limitations that determine whether a given configuration is acceptable or not (i.e. they are constrained). These constrained combinatorial problems can be formally represented as Constraint Satisfaction Problems (CSPs). This provides a language for expressing problems that abstracts away the specifics and instead focusses only on the common elements: the decision variables, their potential assignments, and the constraints upon those assignments. A solution to a CSP can be checked to see if it is acceptable in a small amount of time; however, searching through the vast numbers of potential candidates to find a feasible solution can take exponential time if done in a naïve, brute-force fashion. Solving CSPs efficiently requires a more measured use of computational power.

A number of techniques can be used to solve CSPs; we have chosen to focus on one of the most successful and widespread: the class of iterative-improvement, neighbourhood search algorithms known as Local Search. The design of Local Search algorithms usually incorporates domain-specific knowledge and prior experience. This results in algorithms that perform well in a single setting but are difficult to reuse and transfer to other problems. Alternative solution techniques for CSPs, such as Constraint Programming (CP), do not have such limitations. In CP the search process is entirely independent of the problem definition. The Constraint-Based Local Search (CBLS) movement has arisen in an attempt to bring Local Search more in line with CP. The ultimate goal of CBLS is summarised by the slogan "*Local Search = Model + Search*". CBLS provides a modelling language that allows problems to be captured as generic constraint models; these CP-style models provide information to guide the search component.

1.1 Research Goals and Contributions

This thesis supports the ideals of CBLS and aims to further separate the *Model* and *Search* components. To apply CP to a problem, a user needs to create a model of their problem in the appropriate language; the search is effectively a *black-box* that they do not need to alter. Experienced users, with an understanding of the solver's internal processes, may use their knowledge to find solutions more quickly, however this is not a requirement; CP has a low barrier of entry for novice users. In CBLS the search component remains the user's responsibility; they need to be familiar with the mechanics of the Local Search process and know how to implement neighbourhoods—the functions that define how a search will progress. CBLS still requires prior experience to apply successfully; it is our aim to reduce this prerequisite knowledge and potentially open up CBLS to a wider audience. Reducing the level of human intervention in Local Search introduces the risk of removing the very element that has been integral to Local Search's success—*human intuition.* We want reusable search neighbourhoods for newcomers whilst still allowing more experienced users to supply their own problem-specific neighbourhoods. To compensate for the potential absence of user information the system has to be capable of making its own deductions about the relationships between the search neighbourhoods and a generic constraint model.

The central hypothesis of this work is that relationships between generic constraint models and search neighbourhoods can be detected automatically and subsequently used to inform a search procedure.

For the purposes of this thesis, we focus our investigation on two main questions:

- Can useful information be found from a generic model and search neighbourhood?
- If so, how can this information be used within a search to improve performance?

The first raises further questions:

- How can the CSP problem definition be connected with the search behaviour?
- Will this require any modelling changes?
- What kind of framework is needed to create a generic reusable system?
- How can the behaviour detection be automated?

The second main question will also require several subquestions:

- Can the neighbourhood information be used to choose a neighbourhood to search?

- Do the modelling changes allow for more flexibility?
- Can the information provide the user feedback?

The work in this thesis has two related themes: the extraction of behavioural information, and its subsequent utilisation within a search. The major contribution of this work is the exploration of the relationship between constraints and neighbourhoods, and also the creation of a system for automatically uncovering these connections. This should ultimately reduce the amount of domain-specific knowledge needed to use Local Search to solve CSPs. This will, hopefully, make the adoption of Local Search solutions easier and provide another tool that can be used to more efficiently solve the complex problems. It should also stimulate research into the design of Local Search neighbourhoods. This is an area that, at present, is lacking any real guidelines instead preferring to rely on prior experience and rules of thumb.

Caveats

As well as stating the objectives of this work, it is useful to explicitly set down the caveats. For the information extraction phase of the work, the intention was to produce a workable system that could uncover usable relationships between the model and search. The study of interactions between these components was not meant to be exhaustive. It is possible that there are certain relationships (between constraint models and search neighbourhoods) that our system cannot uncover or classify. There was no intention that this analysis would outperform human analysis; merely that it was no longer a prerequisite. The detection phase is not necessarily the most efficient means of acquiring additional information; its only aim was to prove the concept's technical feasibility.

For the exploitation of the extracted information, the objective was to show *some* of the potential ways that it could be incorporated into a search. We chose to primarily focus on Variable Neighbourhood Search (VNS) style searches, but there could be equally valid applications of this reasoning in other metaheuristics. The hope was that this procedure would be applicable to any problem representable as a CSP; however, it may not provide any benefits for some problems. We envisage it being of most use in problems with a variety of different restrictions; we would not expect it to perform well on problems that have homogenous constraint models. Finally, we did not set out to compete with, or replace, highly-tuned problem-specific algorithms; our intention was always generality over specificity.

1.2 Thesis Outline

This section provides short summaries of the contents of subsequent chapters. Chapter 2 shows the background literature. Chapters 3 and 4 contain the experimental work; each contains its own setup, results and discussion. The final two chapters reflect on the work covered in the preceding chapters; they explore whether it achieves the goals set out in Section 1.1 and where potential future extensions exist.

Background and Related Work

Chapter 2 provides an introduction to the core ideas and summarises the previous work that this thesis builds upon. It starts by introducing the basic *Graph Theory* concepts needed to define what is meant by the term *neighbourhood.* The Travelling-Salesman Problem (TSP) is used as a sample problem to ground the examples. There is some discussion of the *classical* graph search algorithms, tree search, and the properties of search algorithms in general. After tree search, we introduce Local Search and its evolution by looking at its application to the TSP. We outline the elements common to all Local Search algorithms, and explore the strengths and weaknesses of a simple hill-climbing strategy. Metaheuristics arose as a response to the inherent deficiencies of Local Search, so the major algorithms—and their central contributions—are described along with related nature-inspired search techniques. Special prevalence is given to VNS as it forms one of the central components of the work in Chapter 4. The focus of this thesis is on constraint problems, so there is a section introducing CSPs and also the field of CP. After exploring CP, which behaves like the aforementioned tree search algorithms, we cover existing work which is situated in a similar area to our own; the application of Local Search to CSPs. The final section of the chapter outlines the libraries, frameworks and languages that have been developed to promote Local Search usage. By the end of the Background and Related Work the reader should understand what differentiates neighbourhood searches from other graph searches; what constrained problems are, how they have been solved, and how Local Search has been applied to them; and our interpretation of the direction in which the community is moving—towards reusable components exhibiting the problem / search delineation found in CP.

Detecting Constraint-Neighbourhood Interactions

Chapter 3 covers the first main theme of this thesis: automatically detecting the relationships between the neighbourhoods used by a Local Search and the constraints within a CSP. Previous approaches have required either the manual classification of these relationships or the use of a specialised modelling language. The chapter starts with some formal definitions to capture what we mean by *constraint-neighbourhood interactions*. We outline the ways that current modelling practice needs to be altered to achieve these goals; experimental results show that the proposed changes do not adversely affect performance. To create a reusable component we cover the design and some implementation details of a framework in the COMET language. With the necessary architecture in place (to create a non-problem-specific system) we outline a component for uncovering constraint-neighbourhood interactions. For the remainder of the thesis the Timetabling Problem serves as our central example and so we introduce the particular variant used as part of the International Timetabling Competitions (ITCs). A series of experiments show that the Interaction Detector is effective and can find relationships for multiple constraints, neighbourhoods, and a variety of model setups. The first implementation has a weakness for situations where no relationships exist. We enhance the Interaction Detector (via some additional reasoning) to handle these scenarios more efficiently. Finally, we look at the reusability and applicability of interaction information and present a caching system to allow off-line usage of the system. At the conclusion of this chapter the reader will understand what constraint-neighbourhood interactions are, what the Post-enrolment Based Timetabling Problem is, and how structural information about this problem can be extracted automatically with no problem-specific connections.

Exploiting Interaction Information

Continuing from Chapter 3, Chapter 4 investigates how constraint-neighbourhood interaction information could be used during the search process. The first experiment looks at how a VNS could incorporate interaction information into its transitions. The chapter then moves on to the ways that a VNS could use the interactions to create more dynamic neighbourhood ordering that alters depending on the current search state. We introduce a new search algorithm, Constraint Directed Variable Neighbourhood Search (CDVNS), that expands upon the idea of a dynamic VNS. This chapter should have shown the reader some—though by no means all—of the ways that interaction information could be applied during search.

Conclusions and Future Work

The final chapter revisits the major points from Chapters 3 and 4. It connects the principal themes of each chapter with the research goals from Section 1.1. It provides a discussion relating sections of work to the objectives that they achieve. It also comments on the limitations of certain design decisions. By the end of this chapter the reader should be convinced that this thesis has satisfied all the intended outcomes.

A thesis cannot hope to cover every aspect or implication of an idea and as such the final chapter sets out some of the directions that could warrant continued investigation. Neighbourhoods are a central element of this thesis, but we do not claim to provide any guidance for creating these functions. We treat them as black-boxes that can be analysed and exploited automatically; it would be interesting to explore the the automatic creation of neighbourhood functions. Similarly, we would like to look at groups of neighbourhoods and whether collections of neighbourhoods could treated algebraically. The Interaction Detector from Chapter 3 has some limitations regarding completeness and it would be desirable to investigate whether other complete techniques could be applied to overcome these restrictions. Finally, we look at some preliminary work that tries, using constraint-neighbourhood interactions, to automatically partition a multi-phase search.

Bibliography

The bibliography lists the full references for all the citations made in this thesis. Wherever possible the Digital Object Identifier (DOI) for each reference is provided. DOIs are a way of uniquely identifying documents on the internet. Unlike Uniform Resource Locators (URLs) which can change, or become broken, DOIs provide a permanent static reference to a document. To resolve a DOI visit `http://www.doi.org`. In the electronic version of this thesis all the DOI links are active and can be clicked to open up a web browser and take you directly to the referenced document. Some references will require access to online collections such as SpringerLink.

Appendices

Neighbourhoods Summary

This appendix provides detailed information about the neighbourhoods used in the experiments from Chapters 3 & 4. It starts with a description of the seven basic types that any subsequent neighbourhood must extend. These abstract neighbourhoods define what type of moves can be represented within our system and supply many of the underlying methods needed to create a usable neighbourhood. The second section looks at the concrete neighbourhoods; these are the classes that can be instantiated and used within a search. Some of the concrete neighbourhoods are termed *generics*; that is they contain no references to any problem-specific information. The remaining concrete neighbourhoods are connected to the timetabling application used in Chapters 3 & 4. The final part of the appendix describes the role of candidate lists and how they can be used at run-time to customise neighbourhoods' behaviours.

Additional Data

The final appendix contains extra information relating to the experiments in Chapters 3 and 4. Results are predominantly conveyed as graphs, however, there are certain situations where a table may prove more convenient. In cases where the tabular data was deemed too large or too disruptive to the flow of the text it appears in this appendix. All the discussion and analysis of the data remains in the main chapters.

Glossary

The glossary provides short definitions for many of the technical terms and abbreviations used throughout this thesis. The electronic edition has active links allowing the reader to select any unfamiliar term or acronym and jump to its definition in the glossary.

1.3 Usage Notes

This thesis uses a variety of typesetting conventions. References to programmatic concepts such objects, classes or variables are written in monospaced fonts (e.g. **class** `A`); syntax highlighted in a boldface sans serif font indicates the keywords of the language (e.g. **forall** `(i` **in** `1..10)`). Web addresses, the names of specific problem instances, and source code files are displayed as monospaced fonts (e.g. `http://www.strath.ac.uk/cis/`, `~/data/problem01.txt`). Wherever possible the names of particular systems are written in the style used by their original authors (e.g. $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System, COMET, EASYLOCAL++, `EasyGenetic`). British English spellings are used unless referring to a concept, or system, that already has an American English spelling (e.g. taboo vs. Tabu Search). The electronic version of this thesis has links from the code extracts to their original files[1]. This is why the line numbers in the margins of code listings do not usually start from one; they refer to the line number in the source file that the extract is from.

1.4 Publications

During the course of the work covered in this thesis a number of publications were made:

- Alastair Andrew, John Levine, and Derek Long. Constraint Directed Variable Neighbourhood Search. In Yehuda Naveh and Andrea Roli, editors, *Proceedings of the 4th International Workshop on Local Search Techniques in Constraint Satisfaction*, Providence, Rhode Island, USA, September 2007
- Alastair Andrew. Automatically Detecting Neighbourhood Constraint Interactions using Comet. In Kostas Stergiou and Roland Yap, editors, *Proceedings of the CP 2008 Doctoral Programme*, pages 7–12, September 2008
- Alastair Andrew and John Levine. Automatically Detecting Neighbourhood Constraint Interactions using Comet. In Yehuda Naveh and Pierre Flener, editors, *Proceedings of the 5th International Workshop on Local Search Techniques in Constraint Satisfaction*, Sydney, Australia, September 2008
- Alastair Andrew. Exploiting Constraint-Neighbourhood Interactions. In Frank Hutter and Marco A. Montes de Oca, editors, *Proceedings of the Doctoral Symposium on Engineering Stochastic Local Search Algorithms (SLS-DS 09)*, TR/IRIDIA/2009-024, pages 41–45, Université Libre De Bruxelles, Av F. D. Roosevelt 50, CP 194/6, September 2009

[1]The code is hosted online so you will require internet access.

The work presented in Andrew et al. [2007] represents an early iteration of the ideas that evolved into the first half of Chapter 4. Similarly, the work from Andrew [2008] and Andrew and Levine [2008] went on to form Chapter 3.

Chapter 2

Background and Related Work

No one should approach the temple of science with the soul of a money-changer

Sir Thomas Browne (1605–82)
English Physician & Author

The work presented in this thesis investigates how constrained problems can be solved by exploiting the relationships between a problem's constraints and the neighbourhoods explored by Local Search algorithms. This chapter provides the necessary background to understand the contribution of this work. Firstly there is an introduction to the concept of neighbourhoods along with basic graph terminology. To put Local Search algorithms into context, there is a discussion of other traditional graph search algorithms. This is followed by a look at the evolution of Local Search, its main features, and its strengths and weaknesses. The weaknesses of the basic Local Search approach motivated the development of another class of algorithms called metaheuristics; a selection of which are investigated along with other related search strategies such as Genetic Algorithms (GAs) and Ant Colony Optimisation (ACO). The penultimate section of this chapter will cover CSPs, CP and how there has been an increasing movement to combine Local Search with the latter. Finally there will be an overview of some of the software systems that have been created to ease the implementation and adoption of Local Search.

2.1 What *are* Neighbourhoods?

Everyone will have encountered the concepts of *neighbourhoods* and *neighbours* in their day-to-day lives. These terms are usually associated with geographical areas and human society. Our neighbours are the people who stay nearby us and our neighbourhood is the area which surrounds where we live. While we are all familiar with these terms they contain a lot of ambiguity; does a friend who stays two streets away qualify as a neighbour? What defines a neighbourhood? Is it an area with the same post code or telephone prefix? As is so often the case with natural language there are no right or wrong answers to these questions—they can be interpreted in multiple ways. To achieve definitions of *neighbourhood* and *neighbour* which are unambiguous we must instead describe them in mathematical terms. The particular branch of mathematics which can be applied is called *Graph Theory*.

In 1736 Swiss mathematician Leonhard Euler laid the foundations for what was to become *Graph Theory*. At the time he was residing in the city of Königsberg in Prussia[1]. Two branches of the Pregel River[2] meet in the town centre where they create three distinct areas and surround a small island, Kniephof. Fig. 2.1 shows the layout of Königsberg and its seven bridges as they would have appeared in Euler's day. According to legend, one of the popular pastimes for the Königsberg locals was hypothesising about whether it was possible to walk around the town crossing each of the bridges only once. The story may be apocryphal but what is certain is that this puzzle attracted Euler's interest and he set about trying to find a solution. Euler proved that for the bridge configuration of Königsberg it was impossible in his paper *"Solutio problematis ad geometriam situs pertinentis"* Euler [1736][3], but more importantly, in doing so he provided a new mathematical notation for expressing the problem concisely and unambiguously.

In his solution each of the land masses became represented by a single point known as a *vertex*. The bridges between the sections of town were abstracted into *edges* that connected vertices. The resulting structure is a graph and can be drawn as shown in Fig. 2.2. This graph conveys the same information as Fig. 2.1 but no longer retains the same layout; the graph is described in terms of its connections rather than the layout. Instead of representing graphs pictorially they can be concisely written as sets of vertices and edges. A set is a mathematical

[1]Now Kaliningrad, Russia

[2]Russian: Pregoyla

[3]An English translation of the original Latin text appears in Biggs et al. [1976]

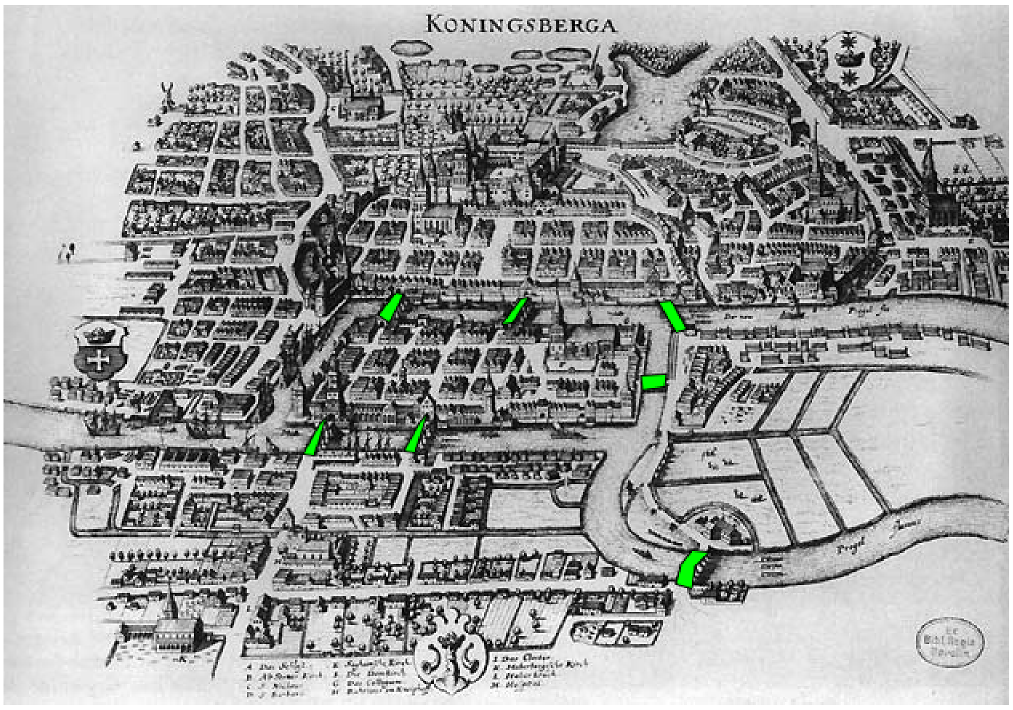


Figure 2.1: Map of Königsberg in 1652 by Merian-Erben. The bridges are highlighted in green.

collection of distinct objects. The number of items in a set is its *cardinality*, conventionally denoted by surrounding the set's name with vertical bars on either side. Membership of an object within a set uses the symbol $\in$ and an empty set can be indicated using $\emptyset$.

Definition 1 *Formally a graph can be described as follows:*

- *A graph, G, is comprised of two sets: one of vertices, $V(G)$, and one of edges, $E(G)$.*
- *A valid graph must have at least one vertex, $|V(G)| \geq 1$ or $V(G) \neq \emptyset$; however, the edge set may be empty, $|E(G)| \geq 0$.*
- *An edge, $e = \{j, k\}$, is a connection between two vertices, j and k.*
- *Vertices connected by an edge are described as being adjacent.*
- $adjacent(j, k) \iff \exists\, e \in E(G)$ *where* $e = \{j, k\}$.

Sometimes it is desirable to use graphs to capture scenarios where relationships only exist in a single direction, e.g. a one-way traffic system within a city. In this case another type of graph, the *directed graph*, is used.

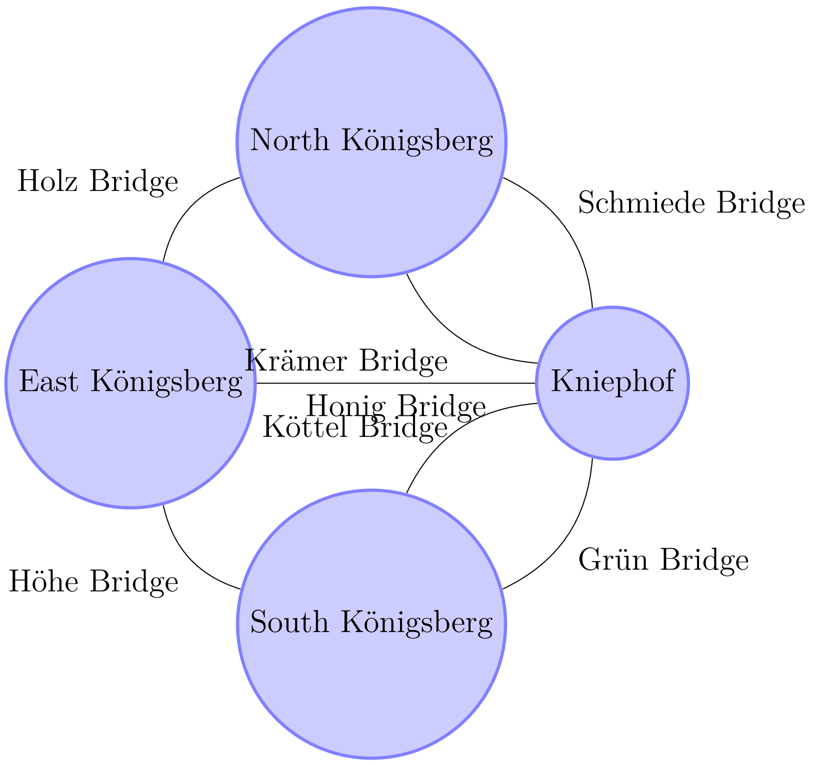


Figure 2.2: Graph of the Königsberg bridge layout.

Definition 2 *A directed graph (or digraph) is defined as:*

- *A digraph, D, is almost identical to an undirected graph except that it has a set of arcs, $E(D)$, instead of edges.*
- *An arc, $a = \langle j, k \rangle$, is a directed edge between two vertices, oriented from j to k.*

Definition 3 *Given the Definitions 1 and 2 we can now formally describe neighbourhoods and neighbours:*

- *Let a vertex $v \in V(G)$ have the* neighbourhood *$N(v)$ which is the set of all vertices that are adjacent to v.*
- *$N(v) = \{x \in V(G) : x \neq v \ \wedge \ adjacent(x, v)\}$.*
- *A vertex n where $n \in N(v)$ is called a neighbour of v.*

Given that we now have mathematical definitions for *graphs* and *neighbourhoods*, how can these actually be used? In Euler's solution the vertices replaced the areas of Königsberg and the edges were the bridges between them. Graphs provide

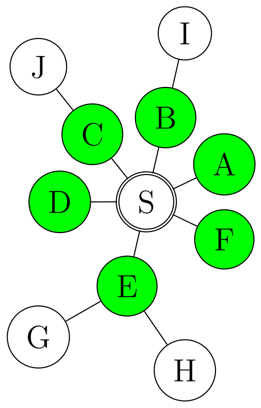


Figure 2.3: A graph highlighting the neighbourhood of S, $N(S)$, in green.

a language with which we can express a large variety of physical and abstract relationships, e.g. telephone network configurations, social circles, and scientific collaborations etc.

2.2 Operations Research

Operations Research (OR)[1] is a branch of applied mathematics concerned with solving real-world optimisation and logistics problems. The advent of World War II focused many mathematicians' research onto practical topics. Cryptography and the "code-breakers" at Bletchley Park often get the better part of the recognition, however numerous other equally important problems were tackled; finding the most efficient configurations for the Atlantic supply convoys and coordinating anti-aircraft batteries to name just two. After the war the solution techniques arising from this research were applied to problems which appeared in civilian industrial settings and in the late 1940s and early 1950s OR started to take off as a serious research field. The placement of chips on circuit boards, scheduling of manufacturing processes, rostering staff, routing deliveries and many other problems fall within the field of OR. These still garner significant research effort because they remain crucial for businesses; anywhere that time, labour or resource consumption can be reduced provides businesses with the scope for real cost savings.

[1]In Europe Operations Research (OR) is called Operational Research.

The Travelling-Salesman Problem

The Travelling-Salesman Problem (TSP) is one of OR's classic optimisation problems, if not *the* classic optimisation problem. The premise is that a door-to-door salesman needs to visit a series of cities. He must visit each only once and return to his starting point. This style of cyclical graph traversal is known as a *tour*. The objective of the TSP is to find a tour which visits all the cities in the shortest distance possible. In *Graph Theory* the term for a tour which visits all the vertices only once (except the start and end) is called a *Hamiltonian cycle*. It is similar to the Königsberg Bridge problem discussed in Section 2.1 but rather than being a decision problem—where the task is to determine whether or not all bridges can be crossed—it is an optimisation problem. Definition 4 shows the TSP expressed formally as an optimisation problem. The difference between decision problems and optimisation problems is minimal. A decision problem can always be converted into an optimisation problem by rephrasing it from *"Does a solution exist?"* to *"Does a solution exist with an $f(x) < k$?"* where $f(x)$ is an objective function being minimised and k is the previous best value.

Definition 4 *Given a set of cities $C = \{c_1, c_2, c_3, \ldots, c_n\}$ and a matrix of* distances $D[n, n]$ *the objective is:*

$$\textit{minimise} \sum_{k=1}^{n-1} D[c_k, c_{k+1}] \tag{2.1}$$

Far from being an academic curio, the TSP occurs in numerous real-world applications. The most obvious scenario is of a logistics company delivering packages to customers, but it has been applied to other diverse problems such as circuit design, robotics control, and Internet Protocol (IP) packet routing. The challenge of combinatorial optimisation problems is handling the growth of alternatives: for a TSP problem with n cities there are *n factorial*, $n!$, possible configurations to consider. The evolution of the TSP as well as Combinatorial Optimisation is dealt with in Schrijver [2005]. A thorough covering of the development of the TSP and related solution methods can be found in Johnson and McGeoch [1997].

2.3 Graph Search Algorithms

When trying to solve combinatorial problems, like the TSP, there are a variety of approaches. The simplest technique would be to perform an exhaustive search by generating every possible combination of assignments and assessing the resultant solutions' quality and feasibility. Visiting every option means that this strategy is *complete*. Completeness is an important property for search algorithms; it guarantees that a solution will be found—if one exists—and, for decision problems, failure to find a solution is irrefutable proof that the problem was not solvable. As we mentioned in Section 2.2 for the TSP this would result in $n!$ states being explored. Consider a 10 city instance: if we wanted to explore an 11 city problem (i.e. only a 10% size increase) the resulting factorial number of states would be $11 \times 10!$ (i.e. a 1000% increase). This rapid growth relative to small input increases is described as the *combinatorial explosion*. It can cause problems to quickly become unsolvable within useful time scales. They are *computationally intractable*. Aside from the time taken to evaluate every possible alternative, storing all the solutions in a naïve fashion would require factorial space in memory. To ensure completeness it is not necessary to store the enumerated solutions, though a systematic scheme of search exploration is required. Treating the search progress as a *tree* allows to solutions to be condensed into paths within the tree. A tree is a graph which has the properties described in Definition 5. Various metaphors are used to when describing trees; some making the analogy with physical trees (e.g. the root node, branches, leaves, etc.) and some drawn from the abstract notion of a *family tree* (e.g. parents, children, siblings).

Uninformed Strategies explore the graph in a methodical manner by performing tree search. A tree is a directed graph that contains no cycles. One vertex is designated as the root node. Any adjacent vertices are known as *children* of a node, it being known as their *parent*. Any node that has no children is a *leaf* node. The depth of a node refers to how many edges away it is from the root.

Definition 5 *A tree is a specific type of digraph with the following characteristics:*

- *the root node is a vertex which has no incoming edges* $\nexists e \in E(G) : e_k = root$.
- *the adjacent vertices to a node are its* children.
- *vertices with no outgoing edges are called leaf nodes, i.e. they occur at the end of the branches in a tree.*
- *the depth, d, of a node is how many steps it is from the root node.*
- *the branching factor, b, is the number of children from a given node.*

- *a tree contains no cycles; that is, there is only a single path from the root to each node.*

Whilst a tree provides a concise structure to capture the search process it does not give any guidance on how the search should be conducted. Tree traversal and the more general Graph Search algorithms define the order the trees and graphs are explored in. They can be split into two main categories: uninformed and informed (or heuristic).

2.3.1 Uninformed Strategies

Breadth-First Search

Breadth-First Search (BFS) Moore [1959] operates by expanding all the nodes at the same depth in the tree before exploring any deeper. Fig. 2.4a shows the order that BFS will visit the nodes in a simple tree. A pleasing property of BFS is that it is optimal for unweighted graphs (e.g. ones where all decision branches have the same cost). That is to say, BFS will find the solution at the earliest point in the tree. There are downsides, though; BFS must keep track of which vertices have been visited and which still require consideration. In the worst case this can result in a space complexity of $O(b^d)$ when b is the branching factor and d is the layer of the tree where the optimal solution occurs. The time complexity is also $O(b^d)$, so whilst BFS is optimal and complete it is unlikely to be the most efficient algorithm applicable.

Depth-First Search

Depth-First Search (DFS) [Russell and Norvig, 2003, p. 75] behaves in a slightly different fashion and instead of exploring each layer, it expands a chain of children until no successor can be found. The search then *backtracks*, retracing its steps until it reaches a parent which still has unexplored children. Fig. 2.4b shows the order in which DFS explores a tree. This results in a time complexity, which is potentially worse than BFS, of $O(b^m)$, where m is the maximum depth of the tree, but a much improved linear space complexity of only $O(bm)$. DFS fails to perform when searching an infinitely large tree. To compensate for this weakness, a common variant called Iterative Deepening Depth-First Search (IDDFS) [Russell and Norvig, 2003, p. 78] has been developed. It works by performing a DFS up to a specified depth limit. If no solution is found then the limit is increased and the search restarts. There is an upper bound after which the whole search will terminate. IDDFS is complete on finite graphs and (as with BFS) will find the optimal solution on unweighted graphs.

Dijkstra's Algorithm

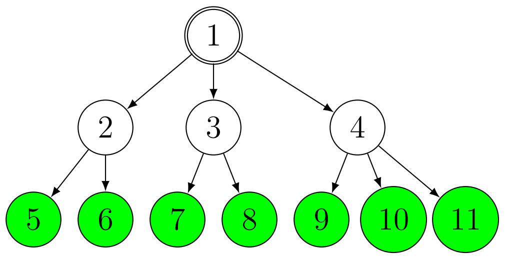
The eponymous algorithm by Dijkstra [1959] is one of the most well-known uniform-cost search algorithms. Whereas BFS, DFS and their variants explore nodes without reference to any cost value, Dijkstra's algorithm adds the notion of distance. This allows it to find the best solution in a weighted graph. Rather than following a preset exploration schedule, it selects the nearest node to expand at each step. The distance to each node, denoted $g(x)$, is updated to reflect the shortest known path found so far. Once all the neighbours of a node have been found it is never reassessed. Dijkstra's algorithm is capable of finding the shortest path to a goal state; however, the distance metric does not provide an indication of which nodes are likely to be nearer the goal, only which are nearer the start.

2.3.2 Heuristic Strategies

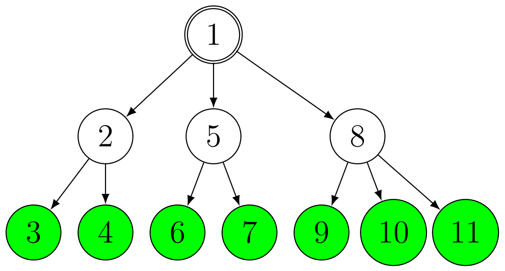
The Oxford English Dictionary, Soanes [2002], definition for *heuristic* is:

> **heuristic** /hyuu-uh-**riss**-tik/ • **adj.** allowing a person to discover or learn something for themselves.
> - ORIGIN Greek *heuriskein* 'to find'.

A heuristic in the context of a search algorithm does not allow learning or discovery per se but does provide the search with additional information for deciding upon a promising direction. Heuristics are a way of differentiating between nodes for expansion. In the uniformed strategies, all the potential nodes are equivalent; the order of discovery determines the order of exploration. High branching factors, or infinite tree depths, can put many problems beyond BFS, DFS, and IDDFS. This does not mean that all is lost; just that exhaustive approaches will struggle. Using additional information in the form of heuristics allows algorithms to prioritise their search effort and hopefully find solutions faster. The trade-off is an increased investment in time computing heuristic scores to—hopefully—produce a shorter overall search run-time.



(a) The order of node exploration made by BFS.



(b) The order of node exploration made by DFS.

Figure 2.4: Examples of the different tree traversals made by BFS and DFS. The root node is shown with a double circle and the leaf nodes are shaded green.

Best-First Search

Best-First Search [Russell and Norvig, 2003, pp. 94–95] takes a greedy approach to node expansion. It uses a heuristic function, $h(n)$, to assign each potential successor node a score, then chooses the *best* score for expansion. The heuristic will depend upon the particular problem being solved, as will the definition of what qualifies as the *best* score. The behaviour is similar to a DFS because the search will continue following child nodes if they provide a heuristic improvement. When it reaches a leaf node (which is not the goal) then it will backtrack and revise an earlier decision. Typically all states will eventually be visited; it is possible to sacrifice completeness by discarding heuristically poor candidates (rather than delaying their exploration until later). To provide useful information for an algorithm a heuristic has to encapsulate some problem-specific knowledge. As a result, heuristics are tightly coupled to particular applications and the search effectiveness depends upon the heuristic's accuracy. If a heuristic can be guaranteed never to overestimate the true distance to the goal state then it is described as being *admissible*.

A* (A-star)

The A* algorithm of Hart et al. [1968] expands upon Dijkstra's algorithm with the addition of a heuristic distance-to-goal estimate. The heuristic function, $f(x)$, in A* is composed of the addition of two sections: $g(x)$, the distance required to reach the current node (which is the same as Dijkstra's) and $h(x)$, the estimate of the distance to the goal state. If the heuristic is admissible then A* is optimal. Also A* can be converted into Dijkstra's by returning the same $h(x)$ for every node. This effectively removes any guidance meaning the search has no way to decide between future directions.

Branch & Bound

In the OR terminology A* is described as a form of Branch & Bound search. This approach has two main concepts: the decision points (where the search branches), and a pruning function which it uses to avoid provably useless subtrees. If the cost to reach a node is greater than a previously seen node's cost plus its distance to the goal, then that path can be pruned. Regardless of how far the node is to the goal the path is already longer than a point that has been encountered. By retaining bounds and pruning infeasible branches the search can be made more efficient whilst retaining completeness.

Graph search algorithms appear in the undergraduate syllabuses of Computer Science degrees across the globe. Their ubiquity is testament to their importance for practical problem solving. Search algorithms have been well studied for decades and can be applied to numerous problems. Self-styled *algorist*[1] and author of "*The Algorithm Design Manual*" Professor Steven Skiena volunteers the advice "*design graphs, not algorithms*" Skiena [2008, p. 222]. By this he means when faced with a problem rather than trying to come up with a completely new algorithm it is often more fruitful to cast the problem into a form that an existing technique can be used with. For example, BFS can be used to find all the connected components of a graph or decide whether it is two-colourable.

2.3.3 Algorithm Properties

We have already seen that completeness is an important property for search algorithms. Another is whether their behaviour is deterministic or stochastic. An

[1]an algorithm designer

algorithm which is deterministic will give the same output for the same state each time. In some situations this is desirable (e.g. when implementing a mathematical function it seems sensible to always return the same answer for a given input). For search algorithms determinism is not always such a useful property. A fixed order of exploration can force the search into repeatedly making poor decisions. Algorithms which make some random choices are *stochastic*. Making stochastic choices reduces the likelihood of the same poor decisions being made repeatedly.

2.3.4 Approximation Algorithms

There are several approaches to deal with the combinatorial explosion and subsequent intractability issues. The simplest would be to give up, admit defeat, and declare that some problems are just too difficult; where would that leave science? A second option is to slacken our definition of a solution; approximation algorithms sacrifice optimality for speed. For optimisation problems, searching for the optimal solution in a useful time scale may be infeasible—but searching for high quality solutions may be achievable. This trade-off is not always possible. Consider a decision problem: either a valid solution has been found, or it has not. Individual circumstances will dictate how appropriate using an approximation algorithm is. In many everyday situations a near optimal solution delivered quickly will suffice. The downside of approximation algorithms is that, unlike the general Graph Search algorithms, they are only applicable to specific problems. To achieve their performance they rely on problem properties.

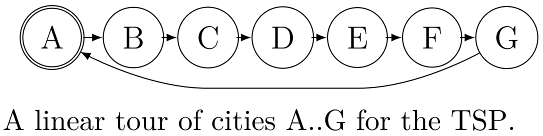
For the TSP a popular approximation strategy is called Nearest Neighbour Next. This constructs a tour by selecting the nearest city which has not already been visited. It will quickly produce reasonable solutions; however, their quality will be dependent on the properties of the particular instance being solved. Another notable approximation algorithm is the First Fit Decreasing (FFD) scheme used for the Bin Packing Problem (BPP). In the BPP there are a series of packages with fixed sizes that must be placed into bins of fixed capacity. The problem is to fit the packages into the bins such that no bin's capacity is exceeded and as few bins as possible are used. The FFD algorithm orders the packages by decreasing size and then attempts to place the largest unassigned object into the first free bin with enough space to accommodate it. This means that the algorithm will only use a new bin when the item will definitely not fit in any of the existing bins. It has been proven that this approach will produce solutions that use—in the worst case—at most $\frac{11}{9}$ $optimal + \frac{6}{9}$ bins Dósa [2007]. If we do not want to preclude the option of finding the optimal solution whilst retaining acceptable run times, then there is a third option: Local Search.

2.4 The Origins of Local Search

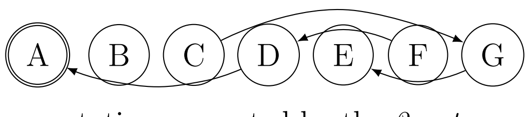
In 1958, G. A. Croes, a researcher working for the Exploration and Production Research Division of the Shell Development Company, submitted a paper to the journal *Operations Research* detailing his approach for tackling the TSP (Croes [1958]). This marked one of the first times that an iterative improvement neighbourhood search was applied to the TSP. The major lasting contribution was the use of the *2-opt* neighbourhood and an iterative improvement strategy. The *2-opt* had in fact appeared nearly three years earlier in a previous *Operations Research* article by Flood [1956]. In the context of the TSP the *2-opt* neighbourhood is a function that permutes an existing solution by swapping the positions of two locations within the current tour to create a selection of successor solutions (the neighbours). Fig. 2.5 shows an example of this move.

All the graph search algorithms encountered so far have operated in a tree search manner, starting from a root node and expanding the search out until a complete solution has been constructed. Iterative improvement algorithms takes a different approach. They start from a complete solution and then attempt to optimise it by making a series of small improvements. The idea that problems can be solved by repeatedly making changes to a candidate solution and then choosing the best improved successor to replace the candidate is the crux of Local Search.

Providing the initial solution was *Hamiltonian* then any neighbours generated by the *2-opt* will also remain so. By only creating valid neighbours the search



(a) A linear tour of cities A..G for the TSP.



(b) A permutation generated by the *2-opt* swapping D and G.

Figure 2.5: Example tours for the TSP.

only evaluates neighbours which could potentially be better. It does not waste time evaluating neighbours that are infeasible. Seven years later Lin expanded upon Croes' work by adapting the *2-opt* into the *3-opt*. Instead of exchanging the position of two cities on the tour now three cities were swapped. It took another few years before Lin and Kernighan expanded Lin's approach by generalising the *3-opt* neighbourhood into the *k-opt* neighbourhood. As the values of k increase there are diminishing search improvement returns versus the increased computation cost to create the neighbourhood (typically a k value will be 3 or 4 at max). The real benefit of the *k-opt* scheme is the flexibility it allows in dynamically switching between schemes.

The success exhibited by Lin and Kernighan on a well-known hard problem which had thwarted other more traditional mathematical solution techniques helped expose and popularise neighbourhood search algorithms. Owing in no small part to its simplicity, the technique arose a number of times and gathered a variety of names: hill-climbing, local-improvement, neighbourhood search, iterated-Lin-Kernighan to name a few. Whilst there are some differences in these algorithms they are all forms of heuristic neighbourhood search and in a bid to simplify and clarify the terminology we will use the generally accepted term Local Search to refer to this class of algorithm.

2.5 Overview of Local Search

Local Search covers two distinct approaches, *perturbative* and *constructive*; both operate with the same basic principle of arriving at an improved solution by making changes to an existing solution. *Perturbative* starts from a fully assigned solution which it attempts to alter by creating a number of slight variations and selecting a new, better solution from amongst them. The perturbative style is the one most commonly associated with the term Local Search, and when we speak about Local Search we mean the perturbative style. The *constructive* style starts from an empty solution and then at each stage assigns a value to the problem variables, growing the solution in a similar manner to the tree searches covered in Section 2.3. A comprehensive exploration of all forms of Local Search can be found in Hoos and Stützle [2005].

Local Search is a delightfully simple algorithm which can come in a variety of *"flavours"* but they all share the same basic ideology and structure (see Algorithm 1).

Algorithm 1: Local Search

```
1 begin
2     $S \longleftarrow$ createInitialSolution()
3     while $\neg$stagnated do
4         for $S' \in$ exploreNeighbourhood($S$) do
5             if acceptanceFunction($S'$) then
6                 $S \longleftarrow S'$
7
```

Initial Solution

The Initial Solution is the candidate set of assignments that the Local Search algorithm permutes in the attempt to find a better solution. There are no restrictions on how this is created. It could be a random assignment of the problem's variables or it may be the result of an approximation algorithm or heuristic scheme.

Neighbourhood

In Section 2.1 we encountered the concept of a neighbourhood in the context of *Graph Theory*. The neighbourhood used within a Local Search algorithm is similar; instead of representing the set of *adjacent* vertices the neighbourhood represents the set of adjacent candidate *solutions*. The term neighbourhood is usually used to refer to both the function which permutes an existing solution to generate the set of neighbouring solutions *and* the resultant set of solutions.

Given that neighbourhoods are central to the operation of Local Search, finding advice on what properties make a *good* neighbourhood is surprisingly difficult. The process has few guidelines or rules that can be followed when implementing an algorithm. This is partially because the neighbourhood is tightly coupled to the problem representation being searched over. Retaining what is known as *connectedness* is usually advised. This means that from any solution there will (given enough applications of the neighbourhood) be a path from the current solution to the optimal configuration.

How the neighbourhood function actually generates the candidate solutions depends upon the problem representation being searched over. The simplest possible neighbourhood is created by changing the value of a single variable to a new value. We describe this as an *atomic neighbourhood* because the permutation it creates cannot be synthesised from any smaller moves. Atomic neighbourhoods can be used as the basis of larger *compound* moves. Other researchers such as Di Gaspero and Schaerf [2003a] and Ågren [2007, p. 33] also promote the idea that neighbourhoods can be *composed* from a series of *atomic* moves. The *2-opt* can be viewed as a compound of two *atomic* assignment operations.

One of the key characteristics of a neighbourhood is its size, that is, how many neighbouring solutions it is capable of producing from any given point. Some neighbourhoods have a fixed size, others are a function based on the size of the instance being searched. The *2-opt* is quadratic in nature and given n variables generates at most $\frac{n^2-n}{2}$ neighbours (which is equivalent to an $\binom{n}{2}$ binomial choice).

Different problems have given rise to specialised neighbourhoods exploiting structural properties which have been later generalised to different applications. One example of this is the Kempe Chain neighbourhood. It started out being applied to Graph Colouring problems but has spread to other related problems like Timetabling. Kempe Chains are a compound neighbourhood strategy in which the neighbourhood size is not fixed, but rather depend upon the configuration being permuted. A Kempe Chain builds a maximally connected subgraph containing all the variables with two distinct value assignments. The permutations can then be generated by swapping the values between the two groups of variables. Another advanced neighbourhood structure, described by Glover [1996], are Ejection Chains. They can be thought of as a sequence of moves in which values are *ejected* from other variables until a stable state is reached.

Some neighbourhood sizes are connected to the problem instance size—as problems become larger, the number of states in the neighbourhoods can grow rapidly. This introduces various problems: chief among them, the traditional strategy of enumerating all the neighbours and selecting between them may either be infeasible or undesirable. A simple way of handling this, which requires close cooperation between the Neighbourhood and Acceptance Functions, involves assessing each neighbour as it is created, ceasing neighbour generation as soon as one is found which meets the Acceptance Function's criteria. Ahuja et al. [2000, 2002] present techniques for handling what they term *very large scale neighbourhoods*. In particular, they highlight how graph solving techniques such as Network Flows can be used to keep neighbourhood exploration within acceptable time limits.

Exploring the neighbourhoods in this linear fashion is not the only option.

Variable Depth Search (VDS) operates by combining single steps from neighbourhoods into one complex move—Lin Kernighan's famed TSP algorithm used this approach. Work by Mautor [2002] proposes the concept of *Intensification Neighbourhoods* which are a parameterised VDS scheme. A tree of neighbours is created by applying different neighbourhoods to generate a predefined number of children at each step. These neighbourhoods can also incorporate cycle detection to prevent wasteful re-exploration of the same solutions. The Intensification Neighbourhoods concept also incorporates Mautor and Michelon's earlier work in the MIMAUSA neighbourhoods [1997]. Dynasearch by Congram et al. [2002] is another VDS strategy, using Dynamic Programming to explore neighbourhoods efficiently. Dynamic Programming is a strategy for obtaining optimal solutions to problems which can be split into overlapping subproblems and for which the optimal solution is comprised of optimal subproblem solutions.

Acceptance Function

The Acceptance Function has two main roles: assessing the quality of the solutions returned by the neighbourhood and deciding which of these neighbours (if any) should be selected. Assessing the quality of solutions depends upon the specific problem being solved, some of which (such as the TSP) already have predefined objectives.

Delta Calculations

In the field of Computer Architecture there is an accepted design principle: "Make the common case fast" Hennessy and Patterson [2003, p. 39]; the same principle can be applied to Local Search algorithms. The evaluation of neighbours is the most common operation in a Local Search and any efficiency gain made here will lead to a noticeable performance increase. Most neighbours will only differ in relatively minor ways (though the particular neighbourhood function will define this), so it follows that their evaluations will not be vastly different either. In this case recomputing each solution's fitness from scratch seems wasteful. Instead, it would be more efficient to just calculate how much they have changed from the previous solution. This is the principle behind *delta calculations*: only a minimal amount of computation is required to evaluate neighbouring solutions. The snag is that actually implementing this functionality can be challenging. It is tightly coupled to the neighbourhood and problem.

Selection Strategy

The most basic selection strategy is to accept the first solution which improves on the current solution's fitness—this is known as First Improvement or Hill-Climbing. The advantage of First Improvement is that it is fast; it only requires exploration until an improving candidate is found. In the worst case the whole neighbourhood will have been enumerated. The drawback of First Improvement is that it is dependent on the order in which the neighbours are traversed. To implement this strategy there needs to be coupling between the neighbourhood function—it would be inefficient to generate all the neighbours and then assess them. The other most popular alternative to First Improvement is known as Best Improvement. This involves evaluating all the neighbouring solutions and then selecting the one that has the greatest improvement over the current solution. Best Improvement guarantees that the search will select the most improving move. However, it does mean that the neighbourhood needs to be fully explored at each iteration. If the neighbourhood is large this can seriously limit performance. If the search finds itself in a situation where none of the neighbouring solutions are improvements then it is said to have reached a local optima. The question of whether it is better to choose First or Best Improvement depends on a variety of factors such as the problem size and the neighbourhood being searched. Work by Hansen and Mladenović [2006] on the TSP using the *2-opt* strategy gives results indicating that First Improvement is the better choice.

Intensification vs. Diversification

The acceptance function is responsible for balancing two conflicting forces: intensification and diversification. Intensification relates to how aggressively the search moves towards local optima. A Best Improvement acceptance strategy is focused on moving to the optima as quickly as possible. This is not always desirable; the more intense a search is, the more likely it is to become trapped at the nearest local optima. Diversification counteracts the effect of intensification. The more diverse a search is, the more widely it samples from the search space and the more likely it is to find areas of high quality solutions. Diversification can be achieved in several ways: exploring larger neighbourhoods (which allows more of the search space to potentially be visited), and accepting non-improving—or even random—moves.

Search Landscape

In the terminology of Local Search geographical metaphors are used as a way of conceptualising the abstract nature of the search behaviour. Indeed, Hill-Climbing has become synonymous with Local Search in some quarters. The majority of these metaphors are a result of describing the search space as the search (or fitness) *landscape*. The *landscape* is a product of two components: the neighbourhood being explored and the heuristic evaluation of the neighbours. Changing either of these elements can alter the landscape. The landscape is usually depicted as a two-dimensional graph, where the y-axis indicates the fitness of states and the x-axis shows the proximity of neighbouring solutions. The resulting image, such as Fig. 2.6, shows a cross-section of a landscape's topography. The peaks in the landscape are local optima, the highest peak is the global optimum. If the search accepts only improving neighbours then it will become trapped at the optima nearest the starting solution. Plateaux are another important landscape feature that are regularly encountered. These are regions of the search space where all the neighbouring solutions have the same fitness evaluation. For greedy heuristically guided algorithms such as Local Search, plateaux can cause problems: Local Search relies on the changing of the solution fitness gradient. In plateaux there is no heuristic information with which to disambiguate potential successors. Local Search can become trapped exploring plateaux due its memoryless nature. Allowing the acceptance of non-worsening moves lets a search wander through a plateau.

The performance of Local Search depends in part upon the variability or *ruggedness* of the search landscape. Local Search is more suited to landscapes which have smoother profiles. Larger numbers of local optima mean the search is likely to become trapped more quickly. Another contributor to the success of the improvement strategies is the idea of a Fitness-Distance Correlation (FDC). This basically expresses the notion that a solution with a higher fitness will be closer to the optimal solution than a lower fitness solution and that there will be a higher density of local optima near the global optima. Related to the FDC is the idea that the search landscape contains *basins* (of attraction). A basin is the collection of all the neighbours from which following the gradient will lead to the same optima. In the labelling of Fig. 2.6 the image is shown as a maximisation problem; in this case the basins of attraction are actually upwards (their naming seems more intuitive when dealing with minimisation). Depending on which basin the search is in determines which of the optima it can reach. Hoos and Stützle [2005, Chp. 5, p. 203] covers the theoretical and practical aspects of search spaces for Local Search in some depth. They expand upon some more advanced landscape concepts (such as the characterisation of search positions and plateau connection graphs) which are beyond the scope of this thesis.

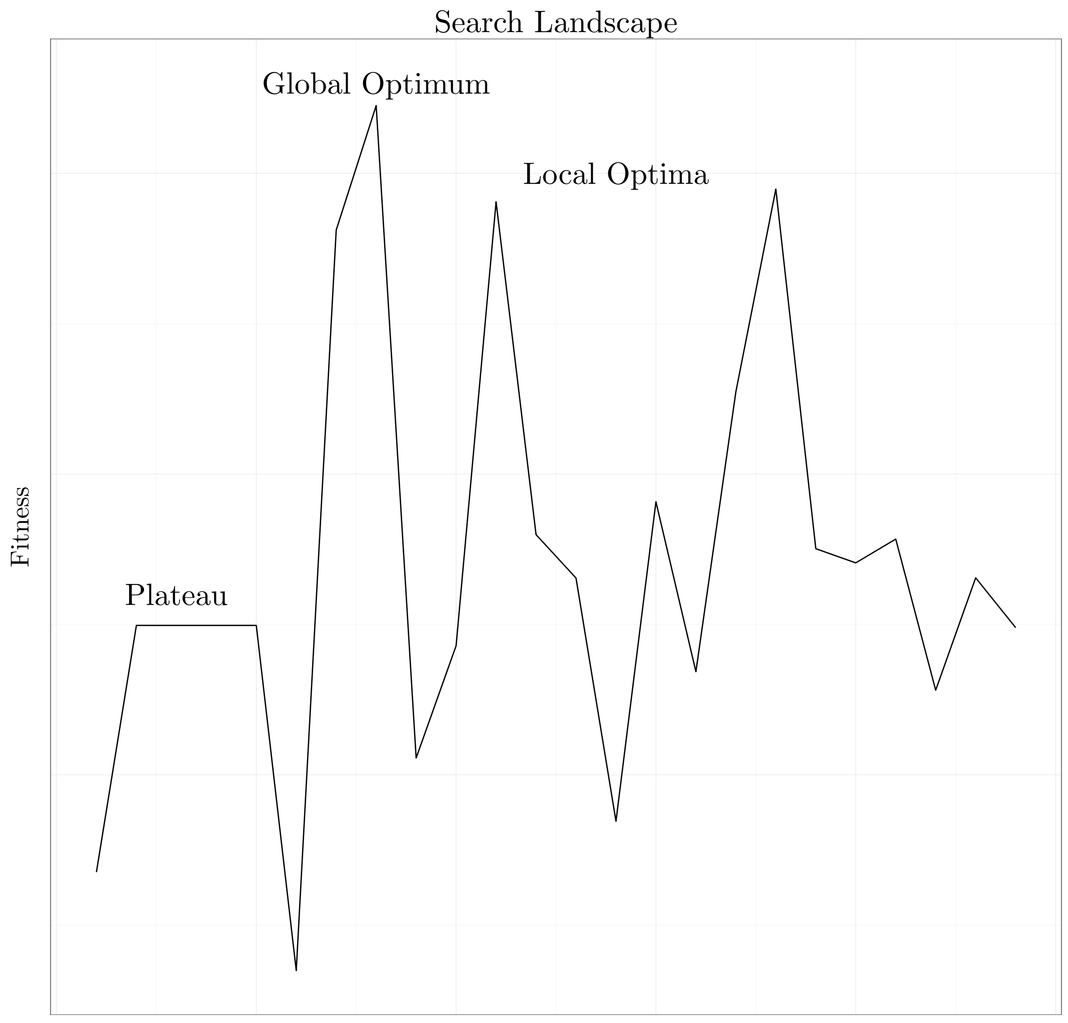


Figure 2.6: An example search landscape highlighting optima (both local and global) as well as a plateau.

2.6 Strengths and Weaknesses of Local Search

Strengths

Simple Structure Perhaps one of the main contributors to the widespread adoption of Local Search (other than its problem solving performance) is its conceptual simplicity. Take a solution, make some alterations; if they improve the solution, keep them. Otherwise, discard them. Iterative improvement is easy to understand and has a compact structure; Algorithm 1 shows the main elements in only a few lines of pseudo code.

Small Memory Usage Because Local Search does not try to retain the entire search tree, it can be applied to problems which are far larger than exhaustive techniques could manage.

Any-time Behaviour As a Local Search progresses it retains the best solution it has encountered so far. This means that the search can be stopped at any point and the best solution encountered so far can be returned. For constructive algorithms the search needs to have been allowed to finish before a complete solution could be returned. The any-time property means that Local Search can be easily combined with other techniques. A specialised construction algorithm could be used to find a solution which is then passed to a Local Search to optimise.

Fast Evaluations By using delta calculations, Local Search can reduce the computational expense of exploring the search space and consequently can visit more solutions than other techniques.

Weaknesses

Of course Local Search is not without its drawbacks. Foremost among these are its tendency to become trapped at local optima. By only selecting improving neighbours, the search intensifies around a single local optima. Most likely this will not be the global optimal. The other main issue is that it is not *complete*. Local Search algorithms traverse the search space by exploring the surrounding neighbours from a particular point and due to their memoryless nature the same solution may be reached multiple times whilst others are never reached. The fact that not all solutions are explored means that Local Search algorithms in their basic form are *incomplete*. Even techniques that are complete will no longer be so if they are halted before they have reached their termination state.

When Local Search algorithms make stochastic choices then they can treated as Probabilistically Approximately Complete (PAC) [Hoos and Stützle, 2005, p. 155]. Although in most practical situations the algorithm will still be incomplete, PAC means that due to the random nature of the search—and given an unlimited amount of time—completeness will be achieved. Work by Fang and Ruml [2004] presented a method of creating a *complete* Local Search (for solving Satisfiability problems). They achieve this by using (in the worst case) an exponential space to store learnt clauses (which act as a form of memory).

2.7 The Rise of Metaheuristics

Beginning in the 1980s and reaching critical mass in the 1990s, a series of algorithmic techniques were developed which became collectively known as metaheuristics. In Section 2.3.2 we encountered the notion of heuristics, which are schemes for estimating the quality of solutions to problems. In keeping with the naming conventions of Computer Science, the prefix *meta* was used to denote that these new techniques were somehow *beyond* and more abstract than problem-specific heuristics. These techniques combine the basic Local Search idea of iterative improvement with a variety of behaviours to overcome some of the weaknesses inherent in a simple first-improvement Local Search.

Iterated Local Search

In Lourenço et al. [2003] they describe a strategy called Iterated Local Search (ILS) which runs multiple Local Searches and uses random disruption to prevent the search becoming trapped at the same local optima repeatedly. The four main components of ILS (as shown in Algorithm 2) are: the Initial Solution, the subsidiary Local Search, the Perturbation Function, and the Acceptance Function. The Initial Solution fulfils exactly the same role as within a normal Local Search. The Local Search in an ILS optimises the current solution until no improvements can be made and it stagnates at a local optima. In a Local Search search the assumption would be that this is the best state available. Unfortunately this is unlikely to be true. Different types of search could replace the subsidiary Local Search. However, the fact that just a simple hill-climbing Local Search will converge quickly on an optima allows for a larger number of ILS iterations.

Algorithm 2: Iterated Local Search

```
begin
    $S \longleftarrow$ createInitialSolution()
    $S \longleftarrow$ localSearch($S$)
    while $canContinue$ do
        $S' \longleftarrow$ perturbationFunction($S$)
        $S^{*\prime} \longleftarrow$ localSearch($S'$)
        if acceptanceFunction($S^{*\prime}$) then
            $S \longleftarrow S^{*\prime}$

        else
            $S \longleftarrow S'$
```

The Perturbation Function takes a solution and disrupts it in some way to create a new solution. The hope is that this perturbation will place the search into a new basin where it can reach new—potentially better—optima. Local Search is applied to this new solution. If the search stagnates at a state better than previously reached then it becomes the point future perturbations are made from. If the state reached proves to be worse than previously encountered the previous solution is restored and another perturbation is made from there. The Acceptance Function within the ILS makes the decision regarding whether the current local optima should be adopted as the best solution.

The form of the disruption or perturbation is important for the effectiveness of the ILS. Too small an alteration to the stagnant solution, and there is a higher chance the search will become trapped at the same optima again. Too large a disruption, and the search will lose the ability to intensify around a suitable optima. Typical schemes for creating this new move are simply selecting a random compound move from a neighbourhood larger than the one being explored by the Local Search component. Any move which cannot be *"undone"* by a single Local Search step is likely to be useful.

Simulated Annealing

When making high quality steel it is important to cool the molten alloy slowly in a series of stages. This allows the atoms within the steel to settle into a crystalline structure mirroring the minimal energy configuration, thus producing a

Algorithm 3: Simulated Annealing

```
1 begin
2     $S \longleftarrow$ createInitialSolution()
3     $T \longleftarrow$ setInitialTemperature()
4     $aS \longleftarrow$ createAnnealingSchedule()
5     while time do
6         for $S' \in$ exploreNeighbourhood($S$) do
7             if acceptanceFunction($S'$) || acceptAtTemperature($S'$,$T$)
              then
8                 $S \longleftarrow S'$
9         $T \longleftarrow$ updateTemperature($aS$)
```

strong and flexible end product. The process is known as *annealing*. Initially the connection between search algorithms and metallurgy may seem fairly incongruous but both are concerned with achieving stable optimal configurations—be that of atoms within steel or value-to-variable assignments. Simulated Annealing (SA) Kirkpatrick et al. [1983] works by introducing an element of randomness into the acceptance function. Improving states are always accepted but as shown in Algorithm 3, non improving solutions can be accepted with a probability.

The key feature is that as the search continues the likelihood of accepting a non-improving solution is decreased, so the search *"settles"* around a single optima. Controlling the decrease of the acceptance probability is the *annealing* or *cooling schedule* which specifies what the *temperature* (which in turn controls the acceptance rate) is at each stage of the search. The actual acceptance criteria at each temperature is based on work by Metropolis et al. [1953] into the simulation of energy configurations within molecules. The Metropolis calculation is an exponential function which is relatively expensive to compute so it is not uncommon to find SA implementations which cache a selection of these values as an optimisation. In addition to defining an annealing schedule, SA requires the setting of an *initial temperature* which determines the starting level of diversification.

Simulated Annealing Variants

A variant of SA called Threshold Accepting (TA) was proposed in Dueck and Scheuer [1990]. Rather than performing the expensive Metropolis calculation, worsening solutions are always accepted provided they are within a certain bound which is decreased over time. The later work of Dueck [1993] adapted the TA algorithm and created two variants: the Great Deluge Algorithm (GDA), and the Record-to-Record Travel (RRT) algorithm. For the GDA they added a *"rain"* parameter which determines by how much the acceptance threshold, or (in their parlance) the *"water level"*, increases after every worsening acceptance. The original TA algorithm required the specification of a full threshold decrease schedule. The *"rain"* parameter removes the need for this. The RRT algorithm is an alternative which replaces their water-based metaphor with the concept of deviation. Any solution whose quality is greater than the best current solution minus a deviation parameter will be accepted. If the quality is better than the best so far found then it replaces the best solution.

The insight behind SA and the related algorithms is that initially they allow diversification but over time this is gradually reduced, forcing the search to intensify around a single (and hopefully global) optima. The initial diversification overcomes Local Search's propensity to become trapped at the nearest local optima.

Tabu Search

Algorithm 4: Tabu Search

```
1 begin
2     $S \longleftarrow$ createInitialSolution()
3     $T \longleftarrow$ initialiseTabuList()
4     while $time$ do
5         for $S' \in$ exploreNeighbourhood($S$) do
6             if acceptanceFunction($S'$) && $\neg$isTabu($S'$) then
7                 $S \longleftarrow S'$
8                 updateTabuList($S$)
```

One of Local Search's strengths is its small memory usage compared with complete tree search algorithms. By dispensing with a complete record of all the solutions explored, Local Search leaves itself open to revisiting states. Tabu Search (TS) Glover [1989, 1990], Glover and Laguna [1997] offers a solution—a short-term memory of recently visited states. As a TS progresses then the solutions are added to a structure known as the *tabu list*. Typically the simplest form of tabu list is implemented as a fixed length queue and as new solutions are added, older solutions are removed. Whilst searching, any neighbours which appear in the tabu list are ignored to prevent fruitless re-exploration. TS chooses the best non-tabu neighbour it can find. It may be that due to the restrictions placed by the contents of the tabu list that this solution may be a worsening step. The number of search iterations solutions are designated as *taboo* for is known as the *tabu tenure*. This introduces design issues of how long the tabu tenure should be; too large becomes prohibitively expensive and negates the small memory advantage of Local Search; too small and the search can unwittingly find itself caught within cycles.

Whilst initially Glover used only a short-term memory, later expansions of TS added a longer-term memory. Remembering a collection of high quality solutions which have been previously encountered is the idea behind the *elite candidate solutions* strategy. The search can choose to restart from a member of the elite pool. However, the previous tabu restrictions are lifted, allowing the search to progress in a different direction than was possible when the state was initially encountered. The elite solutions allow the search to intensify around potentially fruitful areas of the search space. ILS operates like a restricted form of this, having only a single elite solution. Another common feature found in TSs is the *aspiration criterion*, which allows states that may have been designated tabu to still be selected if they will lead to a fitness improvement.

TS is a dissuasion based search where the algorithm is attempting to create a larger diversity by forcing the search towards unexplored areas. The elite candidates mean that when promising regions are found they can be returned to later and hopefully improved upon.

Dynamic Local Search

The term dynamic is used in various contexts when discussing Local Search. In Hertz et al. [1997] they argue that TS can be viewed as a *dynamic neighbourhood* search. For a given state S, the neighbourhood of candidate solutions no longer solely depends upon those returned by the neighbourhood function, $N(S)$, but is also influenced by the contents of the tabu list (which changes during the search process). Dynamic Hill Climbing by de la Maza and Yuret [1994] is an optimisation technique which performs a trajectory based search through a 2-d cartesian space and shares more in common with LP than Local Search. However, neither of these techniques are what are classified as Dynamic Local Search (DLS), which encompasses a range of algorithms who operate by applying weightings to the individual components of solutions in a bid to penalise undesirable aspects.

Probably the most well-known example of a DLS is Guided Local Search (GLS) by Voudouris [1997], Voudouris and Tsang [1995, 2003]. Algorithm 5 shows the structure of the GLS algorithm. When the search reaches a local optima this is with respect to the current objective function, the weightings within the objective function are changed and the Local Search is repeated. The key factor is that the best solution is maintained with respect to the original evaluation function, not subject to any penalty weightings. DLS strategies highlight how artificial and malleable the search landscape is. It need not be a rigid structure; by manipulating its characteristics it is possible to create a topography more amenable to Local Search.

Algorithm 5: Guided Local Search

```
begin
    $S \longleftarrow$ createInitialSolution()
    $P \longleftarrow$ initialisePenalties()
    while $time$ do
        for $S' \in$ exploreNeighbourhood($S$) do
            if acceptanceFunction($S'$, $P$) then
                $S \longleftarrow S'$
            updatePenalties()
```

Adaptive Iterated Construction Search

Greedy Randomised Adaptive Search Procedures (GRASP) by Feo and Resende [1989] is a constructive Local Search algorithm which uses a greedy heuristic to build a solution which is then passed to a Local Search. Each potential assignment is given a heuristic score and the initial constructive phase selects components based upon their heuristic ranking. When it has assembled a complete solution this is used as the starting point for a Local Search. When the Local Search becomes trapped at a local optima, a new candidate solution is constructed and the process begins again. The important part of the GRASP system is that there is an element of randomness in the construction heuristic, it does not always choose the best component; this allows a diverse selection of initial solutions to be constructed.

Starting from an already promising initial solution means that the Local Search component will typically reach a local optima faster than if it were starting from a completely random point. The adaptive element of the search refers to the fact that a component's heuristic evaluation depends upon the existing decisions.

A slight variant on the GRASP idea is that of Adaptive Iterated Construction Search (AICS). In addition to the construction and Local Search phases AICS has an operation which updates the weightings given to the components in the next constructive phase. "Squeaky Wheel" Optimization (SWO) by Joslin and Clements [1999] is an example of an AICS. Possibly inspired by (and certainly named after) the old adage that "*the squeaky wheel gets the grease*", SWO operates by making a greedy constructive solution. The construction section attempts to assign the variables based upon their priority. Once a complete solution is constructed the algorithm identifies variables which it classifies as *trouble makers*. These are the variables that increase the objective function or cause constraints to remain unsatisfied. The priority of these *trouble makers* is increased so that at the next iteration they will be assigned earlier in the construction process.

AICS's interleaving of construction and search is similar to the ILS and TS strategy of retaining elite solutions to restart further Local Searches from. The difference being that rather than uncovering these high quality starting locations during search, AICS algorithms try to explicitly construct them first.

Random Restarts

It is debatable whether random restarts are technically a metaheuristic; they do not manipulate the heuristic function of the Local Search algorithm. However, random restarts are a strategy which can be used to enhance the performance of Local Search algorithms and mitigate against some of the weaknesses, and in that sense they require discussion alongside SA and TS and the other more canonical metaheuristic techniques. Random restarts work as follows: after a given iteration limit has been exceeded (or when the search reaches an optima) the search is restarted. In a constructive Local Search situation the search will start from an empty state. In the more common perturbative Local Search, the search will start from a randomly initialised solution.

This technique has been widely used and occurs even in the early Local Search literature such as Lin [1965]. In Lin's case the restart strategy was being used to try to gain some notion of whether a solution was indeed optimal. The idea was that if from a large number of random runs the search only reached a small set of the solutions then the best of those solutions was likely to be the global optimum.

This does not constitute a proof of optimality but in keeping with the theme of heuristics it offers a rough rule of thumb.

Many of the interesting combinatorial problems exhibit what is known as a heavy-tailed distribution. Informally this means that the longer a search progresses for the more likely it is that it will not find a solution. This property can actually be exploited and in Gomes et al. [1997] they show there is a median point after which the probability of finding a solution by stopping the search and restarting is higher than by continued search. If the search has not terminated by reaching this point then it should be restarted.

Random Walks

As with random restarts, random walks are a strategy or component—rather than fully fledged metaheuristic algorithm—for allowing a GLS to escape from local optima. They first gained widespread attention as part of the WalkSAT (WSAT) algorithm by Selman and Kautz [1993]. Boolean Satisfiability (SAT) is another of the classic $\mathcal{NP}$-complete problems. It was the first problem which was proved to be $\mathcal{NP}$-complete by Cook [1971]. This allowed the complexity of many other problems, such as the TSP, to be established via reduction to SAT. The conventional form of SAT is 3-SAT because it is the simplest variety which the more general $k-$SAT can be converted into whilst also remaining $\mathcal{NP}$-complete.

Definition 6 *SAT can be defined as:*

- *a finite set of variables known as literals, V, $\{x_1, x_2, x_3, \ldots, x_i\}$ where $i \in \mathbb{N}$.*
- *the literals are boolean and can only be either true ($\top$) or $false$ ($\bot$).*
- *literals can be negated, denoted by the $\neg$ operator.*
- *a clause is a disjunction of literals of the form $(x_1 \vee \neg x_2 \vee x_4)$.*
- *the number of literals in the clauses identifies the form of SAT. 3-SAT is most widely used, though 2-SAT[1] and the more general $k-$SAT are also common.*
- *SAT problems are expressed in Conjunctive Normal Form (CNF) which means all the disjunctive clauses are connected by conjunctions e.g. $(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$.*

[1]Note that 2-SAT is no longer $\mathcal{NP}$-complete.

- *the aim of SAT is to find a set of variable assignments such that all the clauses evaluate to true.*
- *a slight variant is MAX-SAT where the goal is to find the maximum number of clauses that can be satisfied.*

One of the earliest Local Search SAT solvers was the GSAT algorithm also by Selman et al. [1992]. For the previous 30 years the backtracking tree search Davis Putnam Logemann Loveland (DPLL) algorithm was the state of the art for SAT solvers; however, GSAT showed that other approaches could be competitive. The GSAT algorithm operates by starting from a random assignment of truth values to the literals and then alternates (or *"flips"*) the literal which leads the largest increase in the number of satisfied clauses. At each iteration it will only change a single literal. Where multiple literals would cause the same increase then it selects one them randomly. There are two main parameters, described as MAX-FLIPS and MAX-TRIES. The former puts an upper bound on the number of assignments that should be tried before restarting the search. The latter limits the number of restarts that occur. GSAT's success paved the way for subsequent Local Search SAT solvers and in particular the WSAT family of algorithms also developed by Selman and Kautz [1993], Selman et al. [1994]. WSAT randomly selects an unsatisfied clause and then chooses either a random variable within that clause to flip or the variable which causes the largest improvement (as in GSAT). WSAT also weights the unsatisfied clauses in a similar fashion to AICS algorithms. Random walks enhance the diversification of a search allowing unrestricted movement through the search space. A random walk is like a less extreme restart. A restart can place the search at any position in the landscape; each random walk step will only alter the solution by a fixed amount.

2.8 Nature-inspired Algorithms

Iterative improvement methods such as Local Search and metaheuristics can be applied to a wide variety of problems, but they are not the only possible algorithms. No discussion of Local Search would be complete without mentioning nature-inspired algorithms. In many respects these techniques can be viewed as forms of Local Search but for historical reasons they tend to be thought of as distinct. So far the techniques covered have focused on improving only a single solution at a time (though there may be multiple elite candidates stored).

Nature-inspired techniques tend to have a pool of solutions from which multiple candidates are being improved at any one time. Having multiple explorations allows a greater diversity to be maintained.

Genetic Algorithms

At the same time as the early groundwork of AI was being researched there was the initial work on Genetic Programming (GP) being performed by the likes of Fogel et al. [1966]. GP attempts to evolve programs by manipulating their structure. This approach is typically applied to languages which have tree structures such as Lisp. However, it was Holland [1975] and his group's work into Genetic Algorithms (GAs) which proved more influential. GAs are a class of evolutionary algorithms directly inspired by the way in which living species adapt to their environments and problems.

The terminology of GAs is strongly influenced by biology—potential solutions are represented by arrays of variables called *genotypes* or *chromosomes*. The individual variables within these *genotypes* are *genes* and the position of a *gene* within the *genotype* is its *locus*. The values which a *gene* can be assigned are called its *alleles*. The canonical form of GAs has binary *alleles* of 0 or 1 but larger domains are possible. Whilst there is a divergence in expressing basic concepts there has also been cross-fertilisation of ideas. Several important developments such as that of the fitness landscape or the FDC covered in Section 2.5.3 were assimilated from the GA field (the former actually being taken from earlier work by biologist Sewell Wright in the 1930s [Mitchell, 1998, p. 8]).

Algorithm 6: Genetic Algorithm

```
begin
    $P \longleftarrow$ createInitialPopulation
    while $time$ do
        $best \longleftarrow$ selectFittestIndividuals($P$)
        $new \longleftarrow$ applyMutationAndCrossover($best$)
        $P \longleftarrow$ replaceIndividuals($P$,$new$)
```

The novelty of GAs is the combination of this simplistic biological representation with a simulated evolutionary process shown in Algorithm 6. Darwinian evolution contained the principles of "*natural selection*" and the so-called "*survival of the fittest*". In a Darwinian sense fitness is measured in terms of success at reproducing—an organism which manages to reproduce is "*fit*". In the context of a GA the fitness of a chromosome is dictated by a fitness function. GAs maintain a population of multiple solutions. To ensure that future populations contain promising solutions, GAs use a mechanism called *selection* to retain a specific number of the top scoring chromosomes.

Where GAs differ from Local Search (other than their increased solution pool) is in their neighbourhoods. New solutions are created by recombining parent solutions using cross-over. Cross-over is the process by which two "*parent*" solutions have part of their chromosomes exchanged to create a new "*child*" solution. A locus within the chromosome is selected and the genes before the locus from one parent replace those in the other parent. More advanced cross-over methods can of course be applied which have multiple cross-over loci. In nature, children are not exact copies of their parents. Random variations in their genes occur. Environmental factors can also alter our DNA, such as exposure to radiation etc. Mutation within GAs is created by the random alteration of genes within a chromosome. This random manipulation fills a similar role to the perturbation functions within an ILS or random walks in WSAT adding diversity to the population.

Evolutionary algorithms are classified using a $\mu + \lambda$ notation where μ is the number of parents and λ is the number of children in the population. The simplest evolutionary strategy is known as a $1 + 1$ strategy. This means that from a population of a single parent there is a single offspring. Exactly the same dynamic is found within Local Search algorithms—a candidate solution being permuted to produce a new solution which can either be selected or not. A good introductory text covering the background and application of GAs can be found in Mitchell [1998]. As well as covering the main features of GAs there is also some comparison of GAs against Local Search techniques. Specifically, Chapter 4 expands upon ideas introduced in an earlier paper by Mitchell et al. [1993] which aims to investigate when a GA will outperform a simple Hill-Climbing Local Search. They highlight that GAs have the advantage of an inherent parallelism due to a population of solutions. However, they make the narrow assumption that a Local Search will only be able to alter a single bit in the solution. Local Searches employ a wide variety of neighbourhoods capable of transitioning between solutions in more complex ways than single bit alterations.

Algorithm 7: Ant Colony Optimization

```
1 begin
2     $P \longleftarrow$ initialisePheromoneMatrix()
3     while $time$ do
4         $S \longleftarrow$ createAntsSolutions()
5         foreach $s \in S$ do
6             $s' \longleftarrow$ runLocalSearch($s$)
7         updatePheromoneMatrix()
```

Ant Colony Optimization

The second nature-inspired algorithm which warrants discussion is Ant Colony Optimisation (ACO), introduced by Dorigo et al. [1991]. ACO takes its inspiration from the way that real ants behave. Biologists noticed that ants are capable of finding efficient paths to food and other important resources. As ants travel they leave a scent trail of pheromones that other ants can detect and follow. In a scavenging situation, the ants will leave their nests and search out food. The ant which returns first will have left the most *intense* trail (since its return journey will have increased the amount of pheromones present). Subsequent ants can choose to either forge off in their own directions or follow the existing (and appealing) trail. The strength of the pheromone trails decay over time and so paths which are not in frequent use become less attractive to other ants.

ACO uses *virtual* ants to construct solutions. A pheromone matrix is connected to the values which the ants assign. During each path construction, the ants can choose either to follow the directions of the pheromone matrix or randomly assign their own values. The amount of pheromone determines how likely an ant is to follow an existing choice. This is a constructive style of Local Search which is closely related to the AICS scheme. The pheromone trails play the same role as the penalty weightings in the AICS search. The main difference is that the penalty weighting within an AICS algorithm does not decay. Algorithm 7 shows the structure of an ACO implementation. More information about the design and application of ACOs can be found in Dorigo and Stützle [2004].

Observations

The metaheuristics and related techniques which we have covered in the previous sections have attempted to address the problem of the search becoming trapped at local optima through several core ideas: accepting non-improving moves, making random moves, starting from multiple regions in the search space. Most commonly, the acceptance function is manipulated so that worsening (or non-improving) solutions can be selected. These worsening steps can either be explicitly selected or the heuristic can be weighted so that they appear to be improving steps. The effect of these manipulations is to make the landscape seem smoother and easier to traverse. Following random moves or randomly disrupting from local optima aids diversification. Rather than trying to explicitly control the nature of the landscape, adding increasing randomness into the search ignores the constraints of the landscape. The addition of memory can both steer the search away from previously visited solutions and suggest the inclusion of desirable attributes. Memory prevents the search from repeatedly making the same mistakes and can be used to prune out attractive yet familiar areas, forcing the search in new directions. Retaining good properties means that search will explore more intensively around existing high quality solutions and, if the FDC assumption holds, other high quality solutions should be nearby. Searching from multiple diverse starting locations can allow a greater coverage of the search space. If the search is too focused on a single area it may never encounter the global optima because the current solution requires too many changes. By having multiple disjoint search locations it reduces the need for any one solution to be vastly altered. These multiple search locations can either be maintained in parallel as a pool or, in the case of ILS, are achieved via the disruptions of the perturbation function.

2.9 Parameterisation

Whilst the metaheuristics and associated technologies have overcome many of the weaknesses that were present in a basic first-improvement Local Search strategy, they have not come without a cost. There has been a large increase in the number of parameters upon which an algorithm's performance depends. ILS requires some stagnation threshold and a perturbation threshold; SA requires an *annealing schedule* and *initial temperature*; TS needs a *tabu list length* and a *tabu tenure*; GLS requires the definition of a weighting scheme; GAs need a *mutation rate*, *population size*, *cross-over likelihood*, etc. In papers describing applications of these techniques the authors will invariably include some discussion about their parameter settings and some justification as to why their particular settings were chosen. This does not mean that these settings are transferable to other problems (or even instances of the same problem). Getting strong performance from these techniques requires a large amount of manual parameter tuning which is tedious at best and introduces the possibility that sub-optimal performance is being reported.

2.9.1 Automatic Algorithm Configuration

The MULTI-TAC system by Minton [1996] was developed to configure the heuristic used in CSP solvers. Choices about what strategy to use for assigning variables or how much pruning to do are defined as parameterised decisions, allowing the system to alter the configuration to solve a given problem. A more general parameter tuning system is F-Race by Birattari et al. [2002] which evaluates potential parameter configurations using a tournament strategy. As with GAs, the population needs to be pruned. If a configuration becomes statistically worse than competing alternatives it is removed. Another approach is the ParamILS family by Hutter et al. [2006, 2009] which tries to tune algorithm parameters via—appropriately enough—a Local Search. The improvement gained after finding the correct parameter configuration for an algorithm can be significant. Hutter et al.'s tuning of the Spear SAT solver took it from being competitive (in terms of number of problems solved) yet slow, to being the state-of-the-art and winning at subsequent SAT competitions.

2.9.2 Reactive Search

The work of Battiti et al. [2008] into Reactive Search takes a different approach to dealing with the increasing parameterisation of algorithms. Rather than trying to optimise a set of parameters offline—as with F-Race and ParamILS— Battiti et al. propose adjusting the search parameters as the search progresses. The search effectively tunes itself to the instance being explored. Battiti has distilled this paradigm over the last decade starting from his work on Reactive Tabu Search in Battiti and Tecchiolli [1994] where the tabu tenure of an item was altered during the search.

2.9.3 Hyper-heuristics

Taking the metaheuristic concept even further results in hyper-heuristics as described in Burke et al. [2003]. If metaheuristics are general solution techniques which can be applied to various problems, then hyper-heuristics are a higher level approach which looks at the problem structure and attempts to determine which metaheuristic strategy will be most successful. In the 2008 AI planner and CSP solver competitions there were strong entries (PbP planner by Gerevini et al. [2009] and the CPHYDRA system of O'Mahony et al. [2008]) using a hyper-heuristic approach; both examples were comprised of a portfolio of lower level solvers. The earlier SATzilla system by Xu et al. [2008] proved similarly successful at the SAT competitions. The scope for this composition is flexible; the hyper-heuristic may simply be a ruleset which is consulted only once for the most appropriate sub-solver for a given problem. Alternatively it may make more complex decisions. The hybridisation may be dynamic with the hyper-heuristic running multiple sub-solvers and allocating appropriate run-times for each one and exchanging solutions between the subcomponents. In Phan et al. [2002] they present a black-box system which combines multiple metaheuristics, Local Searches and exhaustive algorithms in a general framework. During the search the system can decide to change strategy.

2.10 Variable Neighborhood Search

Algorithm 8: Variable Neighborhood Search

```
1  begin
2      $S \longleftarrow$ createInitialSolution()
3      $i \longleftarrow 1$
4      while $time$ do
5          for $S' \in$ exploreNeighbourhood($N_i, S$) do
6              if acceptanceFunction($S'$) then
7                  $S \longleftarrow S'$
8                  $i \longleftarrow 1$
9              else
10                 $i \longleftarrow i + 1$
```

Mladenović and Hansen [1997] were the first to formalise the use of multiple neighbourhoods in a Local Search which they describe as Variable Neighbourhood Search (VNS). Algorithm 8 shows the structure of a classic VNS algorithm. Sometimes the term Variable Neighbourhood Descent (VND) is used to refer to this algorithm, although strictly speaking VND refers to VNSs that immediately return to their first neighbourhood upon finding an improving solution. VNS is often collected in with the metaheuristics, but we have chosen to give it a separate section to highlight that it is not a heuristic (i.e. concerned with the evaluation of fitness functions) so much as a general Local Search framework. Key to the success of VNS are the three observations stated in Hansen and Mladenović [2003, p. 146]:

- a local optimum is only a local optimum with regard to the *current* neighbourhood; it may not be one in a different neighbourhood.
- the global optimal will be a local optimum in *every* neighbourhood.
- local optima in multiple neighbourhoods have been empirically shown to be relatively close (in terms of shared variable assignments).

The approaches covered in Section 2.7 attempt to overcome the Local Search's tendency to become trapped at local optima by manipulating the heuristic evaluation of neighbouring solutions. By making "*worse*" solutions look more attractive they can lead the search away from a purely intensifying behaviour and allow a wider exploration. The random walks strategy is an extreme version of this when no guidance is applied and the search is pure diversification.

From an algorithm design perspective the VNS framework still allows for hybridisation with other metaheuristic algorithms. For instance, no stipulation is made about the type of acceptance function that should be used, and so VNS can be combined with an SA based acceptance function or could be extended to utilise a tabu list or perturbation component. The core simplicity and flexibility of Local Search is retained.

Individual vs. Collective Connectedness

Another advantage of a VNS style configuration is that it removes the need for neighbourhoods to be connected. A connected neighbourhood is one in which it is possible to reach the global optima. If your search algorithm is using a single neighbourhood to traverse the search space then using a connected neighbourhood seems like the most logical choice. However, connected neighbourhoods are typically larger in size and deal in smaller changes than more problem-specific neighbourhoods. Job Shop Scheduling is one application area where the design of neighbourhoods has become increasingly specialised and disconnected, yet high quality solutions are found using Local Search algorithms. In particular the work by Nowicki and Smutnicki [1996] set the benchmark for many years. An empirical investigation by Jain et al. [2000] showed that the neighbourhood employed by Nowicki and Smutnicki (the NS neighbourhood) was the most constrained and explored fewer alternatives than competing algorithms. This tightly focused neighbourhood reduces the exploration of worsening solutions. Though Jain et al. note the NS approach still spends over 99.7 % of its time exploring non-improving neighbours.

If the neighbourhood used can never generate the assignment of values corresponding to the optimal solution then, regardless of the acceptance function, the search will never be able to find the optimal solution. The acceptance function cannot accept a solution that it never encounters. This overlooks the chance that a perturbation function may create the optimal solution, but basing an algorithm on the premise that a random perturbation is required not just for diversification but for completeness seems unwise. Individual neighbourhoods—which are themselves disconnected—can achieve connectedness as a collection within a VNS.

Neighbourhood Ordering

The traditional setup of a VNS algorithm is to have the neighbourhoods ordered linearly by increasing size, the theory being that the most intense neighbourhood should be explored first, and only when the search is unable to find an improvement should a more diverse neighbourhood be explored. The linear ordering ensures that the neighbourhoods explored are the least diverse required (since all tighter neighbourhoods will have been explored first).

VNS provides a framework for the utilisation of multiple neighbourhoods. A more application-specific version of this is often described as a Multi-phase Algorithm. As the name suggests, the algorithms operate in a series of sections using problem dependent decompositions. The initial phase will typically be a constructive component that attempts to find a feasible assignment which provides an acceptable (but suboptimal) solution. The subsequent phases attempt to optimise a function and improve the solution from the first phase. Phases can consist of different types of search but will usually just explore different sets of neighbourhoods.

2.11 Constraints

Constraints represent a set of restrictions or limits that should be respected in valid solutions. Every problem is subject to some form of constraints—in fact the constraints really define what it means to be a problem. Imagine the scenario when you have been asked to provide a solution to a Vehicle Routing Problem (VRP). You have an infinite amount of time, an unlimited number of vehicles at your disposal and no stipulation about the quality of your solution. Does this really constitute a problem at all? A completely random assignment of values to variables would be as valid as one which tries to minimise the time and resources required. Unfortunately, in the real world problems are subject to a large variety of constraints. These constraints make them challenging, but they also provide a set of clues as to the characteristics of an optimal solution.

Constraint Satisfaction Problems

Constraints can be used to capture the structure of a problem in a form known as a Constraint Satisfaction Problem (CSP).

Definition 7 *Formally a CSP is specified as:*

- *a tuple, $CSP = \langle V, D, C \rangle$, comprising of finite sets of variables, V, domains, D, and constraints, C.*
- *a variable, $v_i \in V$, can only be assigned values from its corresponding domain, $d_i \in D$.*
- *variables and domains are typically integer values, though floats, enumerated types, and sets are becoming more common.*
- *constraints specify restrictions over the values that variables can take from their domains.*

The objective is to find an assignment of values to variables so that all the constraints are satisfied. Usually only a single solution is required; however, sometimes all satisfying assignments are desired. Another variant of CSPs are Constraint Optimisation Problems (COPs) which have the same basic structure except that the constraints are split into two categories, C_{hard} and C_{soft}. The hard constraints represent all the constraints which must be satisfied for a solution to be considered valid. The remaining soft constraints form an objective function which must be optimised, as many of the soft constraints as possible should be satisfied.

Constraints come in various forms: a constraint over a single variable is known as unary, e.g. $c_i = (x < 5)$. If a constraint has two variables it is a binary constraint, e.g. $c_i = (x_1 \leq x_3)$. Global constraints can restrict multiple variables.

CSPs are similar to SAT problems but rather than being restricted to just boolean values CSPs allow the variables to take discrete values from a finite set of potential values. This allows for a more compact modelling of problems. For example, imagine the situation where a variable, v, has to take a value from the domain of colours, $D_v = \{red, green, blue\}$. To capture this in a SAT representation requires three variables, v_{red}, v_{green} and v_{blue}, one for each of the possible colour assignments. The first clause needed is one which expresses that at least one of the variables must be true, $(v_{red} \lor v_{green} \lor v_{blue})$. This however is not sufficient because a valid solution would allow more than one variable to be true which does not make sense in a situation when we are trying to choose a single colour. To force only a single variable to be selected, we need to add three additional clauses $(\neg v_{red} \lor \neg v_{green}) \land (\neg v_{red} \lor \neg v_{blue}) \land (\neg v_{green} \lor \neg v_{blue})$. This simple example should hopefully highlight that, whilst SAT solvers are undeniably powerful, the models they use to represent problems can quickly become large and obfuscated. A wide array of CSPs can be found at the CSPLib[1] which serves as a repository for interesting and challenging CSP problems (in much the same way the OR-Library assembled by Beasley [1990] functions in the OR community).

Constraint Programming

The acceptance of CSP problems occured in parallel with the growth of the Constraint Programming (CP) paradigm. Unlike Object Oriented, Imperative, or even Functional Programming, CP is really a technique for solving CSPs rather than a general purpose programming methodology. One of the offshoots of AI was the field of Logic Programming, probably best exemplified by Alain Colmerauer's Prolog language developed in the early 1970s. CP evolved from the Logic Programming community and is still sometimes referred as Constraint Logic Programming (CLP), though this is increasingly rare. The aim of CP is succinctly captured by the belief that "*CP = Search + Inference*". The constraints provide

[1]CSPLib homepage `http://www.csplib.org`

information regarding the potential valid solutions to a CSP. It would therefore seem logical to try to harness this information. It is worth mentioning that in the terminology of the CP community there is special significance attached to the concept of a *solution*. In Local Search a solution is merely an assignment of values to variables—a random assignment is still a solution (though not necessarily a valid or feasible one). The CP community only classifies a complete assignment of values to variables which satisfies all the constraints of the problem as a solution. Throughout the course of this thesis the former convention is used. The search component makes a decision about a variable's assignment and after each decision the inference component attempts to *propagate* the implications of this assignment. The structure which facilitates this propagation is known as the Domain Store. This is an object which maintains the constraints and the domains of the problem variables. The various consistency and propagation algorithms interact solely through the changes they make to the variables' domain. This modular framework allows new constraints to be added extremely easily as there is no interdependence between constraints.

At its core CP is usually a tree-search (like DFS covered in Section 2.3.1) with a number of important additions. Firstly, unlike DFS, CP can use heuristics to choose which variable to assign a value to, and in which order to try the values. Various heuristics exist; First-Fail is one of the most famous: a variable is chosen with the objective of finding an inconsistency as quickly as possible. This may seem counter-intuitive: surely the search should make the decisions which are most likely to find a solution? Actually, it is more useful to know whether a subtree will lead to a solution before too much effort has been invested. First-Fail exploits the second major addition of CP—*backtracking*.

CP has a more advanced notion of backtracking than DFS. In DFS backtracking occurs when the search reaches a leaf node which is not a solution. In CP backtracking can also be triggered at any point by the detection of an inconsistency between the constraints and the remaining values in the domains. CP backtracking is like the search equivalent of an "*undo*" button; rescinding a decision, returning the domains of the variables to their previous state, and (potentially) adding a new constraint enforcing the negated inconsistency. The search progresses by assigning values to variables, whenever the search reaches an inconsistent state, such as a variable having no valid values left, it *backtracks*. The search continues from the last variable which still has unexplored values in its domain. By effectively pruning the search space in this fashion, large subtrees can be avoided as soon as one inconsistency has been found. Merely reactively retracting conflicting assignments does not exploit the full potential that the constraint representation has. The CP community has developed many extensions to backtracking, such as back-jumping (where the search returns to a decision higher in the tree than just the previous decision).

The third major enhancement used within CP solvers is known as *propagation*. CP constraints use propagation algorithms (or just propagators) to remove inconsistent values from their variables' domains. There are several levels of consistency that can be achieved: Node Consistency, Arc Consistency and Generalised Arc Consistency. When an assignment is consistent then every value remaining in a domain will potentially be part of a valid solution. Any values that are not part of a valid solution are pruned.

Node Consistency

Node consistency is the simplest form of inference and is used for unary constraints. It can be performed even before a value assignment has occurred. If we had a variable, x, with the domain $\{1, 2, 3, 4, 5\}$ and a constraint ($x < 3$) then using by node consistency we could prune out $\{3, 4, 5\}$ because none of these values could appear in a valid solution. The propagator would not select a value for x; this would still require a search decision. Only when the domain has a single value left will a propagator make a definite assignment.

Arc Consistency

Arc consistency applies to binary constraints. If we have two variables, x_i and x_j, with the domains $d_i = \{1, 2, 3, 4\}$ and $d_j = \{2, 3, 4, 5\}$ respectively and the constraint, $x_i > x_j$ then there are a number of reductions which we can make automatically: e.g. $\{4, 5\}$ can be dropped from x_j's domain because no assignment of x_i could satisfy the constraint if x_j were assigned those values. A value can remain in the domain of a variable if it has *support* from another value in the other variable's domain. Support indicates that there exists a value in the other domain which would allow the choice of a value to satisfy the constraint. This process of pruning inconsistent values from the domains must be done iteratively to propagate changes through the arc, i.e. a pruned value may have provided support in an earlier constraint which needs to be re-evaluated to ensure it remains consistent. As domains of variables change it triggers events within the CP solver.

The propagation algorithms run until they reach the fix-point where no changes are made (or they encounter an inconsistency and a backtrack occurs).

Generalised Arc Consistency

The final form of consistency is Generalised Arc Consistency (GAC) which provides greater power than just simple Arc Consistency. GAC is also known as Hyper Arc Consistency or Domain Consistency. The classic example highlighting where basic arc consistency fails is a scenario with three variables x, y and z. Each variable has the domain $\{1, 2\}$ and the constraints are $x \neq y$, $y \neq z$ and $x \neq z$. By using just arc consistency this would appear valid because each variable can find a supported value in its neighbours' domains. This is incorrect, since there are only two distinct values and three variables; to spot this flaw all the variables must be considered.

Global Constraints

Global constraints operate on a number of variables and can enforce more complex conditions than can be neatly captured using just unary and binary constraints. Returning to a similar example to the one highlighting the weakness of arc consistency, imagine a situation with three variables, x, y and z. Each variable can be assigned a value from the domain $\{1, 2, 3\}$. This could be modelled as a series of disjunctions to ensure that all the variables take different values (similar to the SAT example in Section 2.11.1), i.e. $(x \neq y)$, $(y \neq z)$ and $(x \neq z)$. However, a more concise and powerful way would be to use the `alldifferent` global constraint to pose this constraint as `alldifferent`(x, y, z). The strength of the `alldifferent` constraint is that it maintains the consistency by phrasing the problem as a bipartite graph matching instance. Global constraints typically have their own propagation algorithms which can be used to efficiently maintain consistency. The availability of global constraints depends largely upon the particular CP system being used; many constraints such as `alldifferent` are ubiquitous. The Global Constraints Catalogue[1] provides a comprehensive listing of constraints and their properties. For the interested reader, a useful introductory guide to CP can be found at Barták [1998]. For a more in-depth treatment of the subject then we would recommend "*Constraint Processing*" by Dechter [2003]

[1]Global Constraints Catalogue homepage: `http://www.emn.fr/x-info/sdemasse/gccat/index.html`

or the earlier standard text "*Foundations of Constraint Satisfaction*" by Tsang [1993].

2.12 Local Search for Constrained Problems

Given that Local Search has been successfully applied to hard combinatorial problems like the TSP, and SAT problems (using the GSAT / WalkSAT family of algorithms), it should come as no surprise that Local Search techniques have also been applied to CSPs and other constrained problems. Even before CSPs and CP had become firmly established. Fox [1983, 1990] was investigating using constraints to guide a heuristic search within a Job Shop Scheduling domain. Minton et al. [1992] provided one of the first applications of Local Search to general CSPs and helped popularise the term *repair method.* The min-conflicts heuristic is a strategy which attempts to *repair* constraint violations by selecting the assignment which causes the most violations, and replacing it with the assignment which minimises the number of violations. This strategy allowed Minton et al. to solve large instances of the $\mathcal{N}$-queens problem several orders of magnitude faster than a backtracking search. Another technique developed at roughly the same time was the *Breakout* method by Morris [1993] which shares many similarities to DLS algorithms. Selman and Kautz acknowledge the resemblance of Morris's work to their own WalkSAT ideas [1993, p. 3].

Despite the success of the *repair* methods, many of the early efforts to combine Local Search and CSPs came from members of the CP community who intended to incorporate Local Search into CP rather than retaining it as a separate entity. Pesant and Gendreau [1996] describe one of first concerted attempts to achieve this by exploring a neighbourhood using a complete Branch & Bound style tree search. Jussien and Lhomme [2002] propose using a constructive form of TS coupled with CP's propagation and consistency techniques; Vasquez et al. [2005] also add consistency algorithms to a TS. Prestwich [2002c] uses *Forward Checking* (which is effectively a propagation algorithm) to enhance neighbourhoods for Graph Colouring. Codognet and Diaz [2001] developed what they describe as an Adaptive Search, which is an extension of Minton et al.'s min-conflicts heuristic. Rather than focusing on the particular assignment which causes the most violations, the Adaptive Search strategy monitors the violations of each of the problem constraints. It tries to apportion "*blame*" onto each variable. The blame of variable is a function of the number of unsatisfied constraints that it appears in. This has parallels with Joslin and Clements's notion of *trouble maker* variables in SWO.

One final comment is that any integration of CP and Local Search must be aware that not all the concepts will be transferable. In a CP model, symmetries in the solution space hinder progress by increasing the number of equivalent states to explore. Research into breaking these symmetries and reducing the amount of redundant exploration is an active topic in the community. Initially, it would seem sensible that performing symmetry breaking in Local Search would yield similar benefits; however, the opposite is actually true. Prestwich and Roli [2005] shows that breaking the symmetries for Local Search can significantly reduce performance. Symmetry breaking makes local optima less distinct and reduces the size of the basin of attraction. They do note that there may be other ways for Local Search to exploit symmetries rather than trying to remove them.

Large Neighbourhood Search

Shaw [1998] developed Large Neighbourhood Search (LNS) which is a remarkably simple hybridisation of the Local Search and CP paradigms. In LNS a complete CP search is used as a neighbourhood within a Local Search to optimise a subpart of the problem. At each iteration elements from an existing solution are removed and then reinserted into the solution using a CP search. The benefit of this approach is that the pruning and consistency techniques of CP can be used whilst the limited size of the reinsertion prevents the tree-based approach from being overwhelmed. Shaw developed LNS for the VRP and it has since proved effective for other constrained problems such as Steel-Mill Slab Scheduling and Nurse Rostering. The idea of using a complete search technique to optimise subcomponents of a problem was also covered by Glover and Laguna, although they describe this process as Referent-Domain Optimization (in Section 10.7 of Glover and Laguna [1997, p. 355]). The MIMAUSA neighbourhoods of Mautor and Michelon [1997] and POPMUSIC by Taillard and Voß [2002] explore a similar approach; solving subsections of a problem with the aim of optimising the complete problem. These are sometimes described as Local Optimisations (LOPT), though the term LNS is more prevalent in the literature.

One aspect that can be somewhat confusing is the origin of the titular term *Large Neighbourhood*, especially as the neighbourhood appears to be a restricted tree search. *Large* refers not to the number of neighbouring solutions it contains but instead to the fact that the moves generated can cause *large* steps through the search space. The reassignment of potentially multiple variables leads to more powerful moves than found in atomic neighbourhoods.

Constraint-Based Local Search

Rather than trying to use a tree-based CP style search, CBLS retains the behaviour of a conventional Local Search. CBLS aspires to CP's elegant decoupling of problem from solver by allowing high level constraint models to be created and searched over using generic Local Search algorithms. COMET is a complete language designed for expressing CBLS algorithms. It evolved from the earlier language, Localizer by Michel and Van Hentenryck [2000], which had formed the basis of Michel's PhD thesis [1998]. Localizer was also a language for creating Local Search algorithms which had a syntax similar to the Optimization Programming Language (OPL)—a previous language for optimisation problems. An overview of OPL can be found in Van Hentenryck [1999]. Localizer transformed into a C++ library called Localizer++ (detailed in Michel and Van Hentenryck [2001]) which ultimately grew into COMET. Initially COMET was solely for expressing Local Search algorithms but this became only one section with current versions supporting CP, LP, Mixed Integer Programming (MIP) and various scheduling specific modules. COMET changed from being a purely academic language into a commercial product marketed by Dynamic Decision Technologies Inc (Dynadec), a company founded by Van Hentenryck. Dynadec closed for business in mid-2013 with COMET's copyright reverting to Brown University. At the time of writing COMET is not available to directly download. As the language has matured the syntax and features have changed, and as a result the "*Constraint-Based Local Search*" book no longer accurately reflects COMET in its present state. Fortunately, an extensive tutorial is distributed amongst the software's documentation (see Dyn [2010]). The syntax of the language looks similar to conventional Object-Oriented Programming (OOP) languages such as C++ or Java. There are some syntactic shortcuts which allow the concise expression of fairly complex constructs.

Differentiable Objects

Although on first inspection CBLS (as exemplified by COMET) appears to use constraints in the same way as CP, this is *not* the case. In CP, constraints (and their propagators) are used to prune the variable domains whilst maintaining consistency. In CBLS constraints perform no pruning, or propagation, and are not used to restrict the assignments explored. Instead, they are used as a form of search heuristic where the number of constraint violations act as an objective function providing guidance to the Local Search. Nareyek [2001] also proposed using global constraints within Local Search for heuristic information, though appears unaware that this was already present within Localizer.

The key feature of constraints within CBLS is their *differentiability*. In Section 2.6.1 we highlighted that delta calculations can substantially boost the performance of Local Search algorithms. In mathematics, differentiating a function calculates the rate of change at a given point. In CBLS, differentiating a constraint returns the delta change of applying a potential assignment. Implementing delta functions correctly is a time consuming and challenging endeavour. COMET provides delta functions for all its built-in constraints and objective functions, significantly reducing the effort required to create efficient Local Search algorithms.

Invariants

Another important feature of CBLS algorithms is the use of invariants. Invariants are incrementally calculated variables that are used to maintain useful information about the search or problem. For example, whilst solving the TSP problem it is important to know what the current variable assignment's tour length is. The tour length itself can be simply stated as being the summation of the distances between all the locations (plus the distance between end and start), as was shown in Definition 4. As the search progresses, and assignments are changed, we want the tour length to reflect these changes; an invariant allows the expression to be stated once and then effectively forgotten about. COMET guarantees the invariant will be correctly maintained during the search and internally uses efficient incremental algorithms to update only the altered sections. This means that invariants are a powerful modelling tool for the algorithm designer, freeing them of handling the low level issues required to maximise efficiency.

Using constraints to guide a Local Search turns out to be an effective way of solving CSPs; Schaus et al. [2011] show that CBLS dominates CP and LNS when solving the Steel Mill Slab Design Problem. The Steel Mill Slab Design Problem arose from a real industrial problem at a steel manufacturer; it is concerned with satisfying a series of orders for different sizes of steel sheets whilst reducing the wastage as much as possible. For many years it provided a challenging benchmark in the CP community (where it appears as `prob038` in the CSPLib). Schaus et al.'s CBLS approach performs so well that they rendered the original instance trivial and were obliged to create a set of 380 new, harder instances.

Constraint-Oriented Neighborhoods

Whilst LNS uses a CP backtracking search as a neighbourhood, there have been other attempts to exploit the connections between conventional neighbourhoods and the problem constraints. The work of Viana et al. [2005] on the Unit Commitment Problem (UCP) proposes the concept of Constraint-Oriented Neighborhoods (CON). This represents the first direct assertion that neighbourhoods can alter the violations of the problem constraints in differing fashions and that this property can be intentionally exploited.

Previously, this had always been implicitly used. For example, the prevalence of the *2-opt* neighbourhood in the TSP literature is due in no small part to the nature of its permutations. The *2-opt* is incapable of creating neighbours which violate the Hamiltonian constraint. Viana et al. motivate their work by citing increasing parameterisation of other applicable algorithms (see Section 2.9). They also highlight that decision makers' reluctance to use metaheuristics for real-world problems is due to the performance relying on obscure parameters and the lack of a rigid mathematical foundation.

They introduce the notion of *hard recovering constraints.* These are problem constraints which, once violated, may be difficult to satisfy again. These hard recovering constraints must be detected manually in a pre-analysis phase. For problems exhibiting these constraints the authors believe that defining more elaborate neighbourhoods, which quickly return the search to feasibility rather than exploring the infeasible space, will lead to a smoother search trajectory. All possible combinations of constraint violations for the *hard recovering constraints* must be considered, with a neighbourhood created for each situation. The same neighbourhoods may be applicable to multiple scenarios, but in the worst case a problem with n hard recovering constraints will require the consideration of an exponential 2^n neighbourhoods.

The structure of the CON is given in Algorithm 9. An initial feasible solution is created via a constructive algorithm after which the search iterates, altering the assignments of one or more variables. If this change has led to an infeasible solution, then the constraint violation state is compared against the pre-computed hard recovering constraint analysis to return an appropriate neighbourhood to hopefully return feasibility. The neighbourhood is explored leading to a new solution which can either be accepted or rejected as the new candidate.

In later work Viana et al. [2008] extend the analysis and run comparisons against various forms of GAs and a GRASP algorithm. The hybridisation of GRASP and CON provides substantially better results for the UCP in a fraction of the time.

Algorithm 9: Constraint-Oriented Neighborhoods

```
1 begin
2     $S \longleftarrow$ createFeasibleInitialSolution()
3     while $\neg stopping$ do
4         changeVariableState($S$)
5         $violatedConstraints \longleftarrow$ checkConstraintViolations($S$)
6         $N \longleftarrow$ selectNeighbourhood($violatedConstraints$)
7         $S' \longleftarrow$ selectNeighbour($N$, $S$)
8         if acceptanceFunction($S$) then
9             $S \longleftarrow S'$
```

Constraint-Directed Neighbourhoods

Fox [1983] introduced the term *Constraint-Directed Search*, where it was used to describe a Beam Search selecting moves in a Job Shop Scheduling problem to resolve constraint violations. This represented one of the first times that the constraints themselves became the central focus of the search. Ågren [2007], Ågren et al. [2009] have developed the concept of Constraint-Directed Neighbourhoods (CDN). In Section 2.11 we saw how constraints could be utilised to infer additional information, in that case by using consistency and propagation techniques. Ågren et al. propose that the constraints themselves can be used to generate sets of potential neighbours to explore. Furthermore the neighbours are partitioned into three sets based upon whether they increase, decrease or maintain the current violations of a constraint. This happens before they are ever evaluated. CDN forms Chapter 6 of Ågren's doctoral work (in earlier work Ågren et al. [2007a] this technique is known as Constraint-Oriented Neighbours, but we shall use CDN to avoid confusion with Viana et al.'s CON).

The main thesis promotes the use of set variables for Local Search. This approach uses Existential Monadic Second-Order logic ($\exists MSO$) to allow the creation of generic differentiable functions for global constraints. Ågren's system has various built-in constraints including `Partition`, `MaxWeightSum`, `AllDisjoint` and `MaxIntersect` for which differentiable implementations are given. However, the flexibility of the $\exists MSO$ approach means that if a required constraint is not present, the incremental penalty and differentiation functions can be automatically generated. Admittedly these might not measure the constraint in exactly the same fashion as a human's implementation, but in terms of reducing effort and increasing the extensibility of the system it is a desirable feature. Problems need to be modelled using set constraints, which is unusual as set variables are not well supported in most CP or Local Search systems.

The generic creation of the CDNs uses five basic moves: `add(S,v)` inserts a value v into set S, `drop(S,u)` removes a value u from set S, `flip(S,u,v)` replaces u in set S with v, `transfer(S,u,T)` removes u from set S and inserts it in set T, and `swap(S,u,v,T)` swaps u from set S with v in set T. The `add` and `drop` neighbourhoods are the atomic neighbourhoods that are combined to form the remaining three moves. The limitation of this representation is that it is incapable of expressing neighbourhoods based upon what they term as *value-directed neighbours* Ågren [2007, p. 170]. This mean that fairly simple concepts such as exchanging a violated value to another variable within a range cannot be captured.

Aside from providing the set formalism required to generate differentiable constraints and neighbourhoods, Ågren explores the generation of Multi-phase algorithms. As covered in Section 2.10, Multi-phase algorithms partition the search into a number of *phases* in which various subsets of the problem constraints are satisfied. Ågren considers only a two-phase approach in which an initial set of constraints are satisfied, and then remain so whilst the rest of the constraints are fixed in the second phase. To identify the constraints to be partitioned into these phases they take an approach similar to those used for automatic parameter tuning in Section 2.9.1. Potential partitionings are generated and the time to solve the problem—in this case the Progressive Party Problem (PPP)—is measured. Ultimately the partition which allows the most instances to be solved in the shortest time is selected. Ågren concedes that it may be possible to generate partitionings via an offline static analysis approach, though this is left as an open research question.

2.13 Assisting Local Search Implementation

As seen in the preceding sections there are a wide variety of algorithms which fall under the general *“umbrella”* of Local Search. These algorithms have been applied to a diverse range of combinatorial, constraint and optimisation problems with much success. Even though one of their greatest strengths is their simplicity and general applicability, this does not easily translate into writing non-problem-specific implementations. End users are invariably forced to recode their algorithms for each new problem they wish to apply Local Search to. Anyone familiar with the software development process may not be that surprised by this lack of reusability in existing code. However, given that CP and LP solvers have managed to retain a clear delineation between their problem model and the actual search techniques, there seems no reason why Local Search should remain so tightly coupled.

In the late 1990s there began a series of projects which aimed to ease the adoption of Local Search and related technologies by providing tools and support for users. These took several forms: primarily as frameworks for existing programming languages such as the C++ and Java; others as libraries as part of alternate systems; a few more were fully fledged languages.

Frameworks

OpenTS a Java TS by Harder [2001][1].

HotFrame a C++ heuristic search framework by Fink and Voß [2002][2].

Searcher an OOP Local Search framework by Andreatta et al. [2002].

EASYLOCAL++ a C++ framework for Local Search by Di Gaspero and Schaerf [2003c].

Compose a C++ CBLS framework containing global constraints (with a particular emphasis on scheduling constraints) by Bohlin [2004]

HSF a Java Heuristic Search Framework (HSF) by Dorne and Voudouris [2004].

[1]OpenTS homepage `http://www.coin-or.org/Ots/`

[2]HotFrame available from `http://www1.uni-hamburg.de/IWI/hotframe/hotframe.html`

HeuristicsLab a C# framework for GA and Local Search development by Wagner and Affenzeller [2005].

MDF the Metaheuristics Development Framework (MDF) a C++ framework for rapid hybridisation of metaheuristics by Lau et al. [2007].

TMF a Templatized Metaheuristics Framework (TMF) by Watson [2007].

METSlib a C++ metaheuristics framework by Maischberger [2009][1] which is part of the Coin-OR project like OpenTS.

or-tools a collection of CP and LP components (with some Local Search support) developed at Google by former ILOG staff[2].

OscaR a Scala framework for CP, LP, MIP with some CBLS features developed by OscaR Team [2012]

Many of the earlier frameworks for developing Local Search algorithms are covered in Fink et al. [2003], though most of the examples focus on the authors' own HotFrame work. Of the frameworks available, EasyLocal++ has had the most sustained development. The project has remained active for the best part of a decade. EasyLocal++'s core is a C++ framework which provides the skeleton functionality for developing Local Search algorithms. Further extensions have included automating the creation of neighbourhood stubs with EasySyn++ by Di Gaspero and Schaerf [2007] and adding features to support the development of GAs with `EasyGenetic` by Benedettini et al. [2009].

Libraries

ECL^{i}PS^e repair The repair library allows the ECL^{i}PS^e project (described in Apt and Wallace [2007]) to support Local Search.

ECL^{i}PS^e tentative largely supersedes the older *repair* library.

ILOG Solver added in version 5.0, ILOG Solver has Local Search features covered in Shaw et al. [2002].

ParadiseEO a C++ framework for metaheuristics and population based Local Search is in the MO library metaheuristics in Cahon et al. [2004].

[1]METSLib homepage `https://projects.coin-or.org/metslib/`
[2]Google's or-tools are available from `https://code.google.com/p/or-tools/`

The libraries add support for Local Search to systems which were not initially designed with Local Search in mind. The ECL^{i}PS^e project (which is a Prolog style language) is one of the longest running CP systems. Since September 2006 ECL^{i}PS^e has been open-source (after its purchase by Cisco Systems). ILOG Solver is the CP counterpart to their MIP optimisation suite, CPLEX. Within the Solver package Local Search support has been added so that it supports LNS as well as more general Local Search algorithms.

Languages

SALSA a language for expressing both Local Search and tree searches using the same constructs by Laburthe and Caseau [2002].

Localizer an OPL style language for the Local Search by Michel and Van Hentenryck [2000].

Comet an OOP language building upon Localizer to support CBLS as well as CP, LP and MIP by Van Hentenryck and Michel [2005]

Of the languages, COMET was most recently active. The initial focus on solely CBLS was broadened to include other search paradigms and more effort was placed on making COMET a robust and usable language rather than solely an academic byproduct. Ultimately the failure of the majority of these systems was not due to any inherent technical problem, but rather the lack of general uptake and the development of any significant user base outside the original authors.

As well as languages, libraries and frameworks there have also been moves to formalise the study of Local Search algorithms through the development of the Generalised Local Search Machine (GLSM) model by Hoos and Stützle [2005, Chp. 3, p. 113]. Hoos and Stützle use representations which are a form of non-deterministic Finite State Machines (FSMs). Motivated by the fact that most high performance Local Search algorithms are actually hybridisations of multiple Local Search and metaheuristic components the GLSM allows these complex structures to be captured succinctly. The states within the GLSM are the Local Search component being explored, and the transitions between states can be conditional or unconditional and can be either probabilistic or deterministic.

2.14 Summary

This chapter has given an overview of various types of combinatorial problems and the techniques which exist to solve them. In particular it has focused on the development of Local Search based solutions. We have covered the structure and properties of Local Search and how the weaknesses of a simplistic first-improvement approach fuelled the creation of the metaheuristic strategies. We gave an overview of the main metaheuristic strategies and saw how they operate by altering the acceptance function criteria of the search in a bid to escape local optima. We also looked at VNS which uses multiple neighbourhoods to diversify the search without manipulating the heuristic values of solutions. We have outlined what CSPs are, their relationship to CP, and how there has been a movement to bring Local Search and CP closer together. Finally we looked at some of the languages and frameworks which have sought to increase the adoption of Local Search.

This move towards a clean CP style model separation raises some interesting issues. Traditionally Local Search has been tightly coupled to the particular problem being solved—can problem information be extracted in a non-problem-specific fashion? Ågren [2007] and Viana et al. [2005] investigate the connection between the problem constraints and the search neighbourhood movements. Both use either specialist representations (such as a complex $\exists MSO$ model) or require human domain analysis to extract the behaviour of neighbourhoods. Can these be incorporated into one of the existing CBLS frameworks so that problem information can be extracted without either complex modelling or user intervention? The next chapter investigates this quandary and presents a method for achieving this using the COMET language.

CHAPTER 3

DETECTING CONSTRAINT-NEIGHBOURHOOD INTERACTIONS

A wise man is strong; yea, a man of knowledge increaseth strength.

Proverbs 24:5
KING JAMES BIBLE

The previous chapter charted the emergence of Local Search from the OR community and its application to a variety of optimisation problems including CSPs. It also covered how Local Search had been adapted to make it more amenable to solving constrained problems starting from early work like the min-conflicts strategy of Minton et al. [1992] through to later advances like CON by Viana et al. [2005] and CDN by Ågren [2007]. Finally, there was an overview of the technologies which aimed to make the adoption of Local Search easier; COMETbeing the most stable and well-supported (though ultimately defunct) of these. Both Viana et al. and Ågren's work require that the search neighbourhoods are annotated with additional information about which of the problem constraints they affect. In Viana et al. this analysis is done manually; Ågren's specialised $\exists MSO$ constraint model implicitly captures some of this interaction information. This chapter covers how this neighbourhood behaviour information can be extracted automatically in COMET without any specialised modelling alterations or human intervention.

The first section of this chapter formalises what we mean by neighbourhood behaviour information and introduces what we term *constraint families* and *constraint-neighbourhood interactions*. These provide a way of capturing the relationships between neighbourhoods and groups of constraints. Maintaining constraint families in COMET requires a slightly different approach to modelling problems. The second section of this chapter details some experiments to ascertain whether these modelling changes are efficient enough to be practicable. The next section provides a discussion of the COMET framework we created to allow flexible models that are decoupled from generic search neighbourhoods. The fourth section describes the Interaction Detector: a problem-independent component for automatically identifying constraint-neighbourhood interactions. Section five introduces the Post Enrolment-based Timetabling Problem which serves as a recurring example throughout the remainder of this thesis. The sixth section contains the experimental evaluation of the Interaction Detector.

3.1 Defining Constraint-Neighbourhood Interactions

First, it is important to clarify exactly what is being detected. To do this we introduce the concept of *families* of constraints. Grouping constraints into related families allows us to study the different effects that neighbourhoods have on them (i.e. the families as a whole). Recall that the definition of a CSP from Section 2.11.1 was the tuple $\langle V, D, C\rangle$. We can expand upon this description using the notation found in Dechter [2003, Sec. 1.3, p. 12].

- In this notation C is the set, $\{c_1, \ldots, c_n\}$, containing all the constraints over the variables in V.
- A relation R is a subset of the Cartesian product of the variables' domains, $R_i \subseteq D_1 \times \cdots \times D_n$. The variables within a relation are known as its scope, $S_i = \{x_1, \ldots, x_n\}$.
- A scope S is a subset of the problem variables V, $S \subseteq V$.
- A constraint c_i is a tuple comprised of a scope S_i and a relation R_i, $c_i =$

$\langle S_i, R_i \rangle$. The relation R_i is the set of partial assignments that satisfy the constraint.

- An *instantiation* of a set of variables is an assignment of a value to each variable in the set, $\{\langle x_{i1}, a_{i1}\rangle, \ldots, \langle x_{ik}, a_{ik}\rangle\}$.
- A *projection* π_a is a set formed by the restriction of the instantiation tuples to just the desired attributes.
- An instantiation $\bar{a}$ satisfies a constraint c_i if and only if (iff) it contains a projection of values which appears in the R_i of the constraint, $\pi_{value}(\bar{a}) \in R_i$.

To illustrate these terms consider a problem with three variables: x_1, x_2, and x_3. The domains for the variables are $D_{x_1} = \{1, 2, 3\}$, $D_{x_2} = \{2, 3, 4\}$, and $D_{x_3} = \{5, 6\}$. There is one constraint $c_1 = x_1 \geq x_2$ comprised of the scope $S_1 = \{x_1, x_2\}$ and relation $R_1 = \{\langle 2, 2\rangle, \langle 3, 2\rangle, \langle 3, 3\rangle\}$. An assignment $\bar{a} = \{\langle x_1, 3\rangle, \langle x_2, 2\rangle, \langle x_3, 6\rangle\}$ projected over the variables in S_1 results in the set $\langle 3, 2\rangle$. This is a satisfying assignment because it occurs within R_1. The value of x_3 does *not* appear in the projection because it is *not* part of the scope of c_1 and is, therefore, irrelevant.

Constraint Families

In COPs C is actually the union of the sets of hard and soft constraints, $C = \mathcal{C}_{\text{hard}} \cup \mathcal{C}_{\text{soft}}$. In our terminology the hard constraints, $\mathcal{C}_{\text{hard}}$, and soft constraints, $\mathcal{C}_{\text{soft}}$, are *families* of constraints. These families are exactly the same as the traditional definition of C except that they only contain a subset of the constraints, (i.e. $\mathcal{C} \subseteq C$). The total number of constraints remains the same regardless of whether a problem is represented as a single family or as multiple families.

Definition 8 *For a problem with n constraint families, $\mathcal{C}$, the set of all constraints, C, can be stated as:*

$$C = \bigcup_{i=1}^{n} \mathcal{C}_i \quad \textit{where } n > 0 \tag{3.1}$$

$$|C| = \sum_{i=1}^{n} |\mathcal{C}_i| \tag{3.2}$$

Definition 9 *The violations of a constraint family $\mathcal{C}$ and the set of all constraint families C for an assignment $\bar{a}$ can be found by the following functions:*

$$violations(\mathcal{C}, \bar{a}) = |\{c_i \in \mathcal{C} : \pi_{S_i}(\bar{a}) \notin R_i\}| \tag{3.3}$$

$$violations(C, \bar{a}) = \sum_{\mathcal{C}_i \in C} violations(\mathcal{C}_i, \bar{a}) \tag{3.4}$$

The problem definition of a COP will explicitly state the family that each constraint belongs to. When trying to define constraint families for CSPs it becomes more challenging as there is no guidance. To provide an example to ground this discussion we shall consider the Progressive Party Problem (PPP) which appears as a COMET model in Michel and Van Hentenryck [2002] and Van Hentenryck and Michel [2005, p. 185], a *Composer* model in Bohlin [2004] and as set-based model in Ågren [2007], Ågren et al. [2005, 2007a,b, 2009]. The PPP was first studied by Brailsford et al. [1996] and Smith et al. [1996] where it was used to compare the modelling power of Integer Programming (IP) and CP. The problem revolves around scheduling a social event at a yacht club. There are 39 yachts, each with between one and seven crew members on-board. The idea is that some of the yachts will be hosts and hold parties for the other crews. The event is split into six half hour periods during which the crews of the host boats remain on their own yachts to cater for the other visiting guest crews. To make the gathering interesting the guests should only visit a host boat once and they should meet any other group of guests at most once. The guest crews move around as inseparable units all evening. The size of the yachts constrains how many guests and hosts can be accommodated on each vessel. The version of the PPP appearing in Smith et al. [1996] is phrased as an optimisation problem where the aim is to reduce the number of hosts needed. Our formulation is closer to that of Van Hentenryck and Michel [2005] in that it is cast as a decision problem; can the event be scheduled with a fixed number of hosts? To model the problem the four required constraints are:

c_1 Each guest boat must attend a party at every period.

c_2 The capacity of each boat must be respected.

c_3 Each guest has a unique host for every period.

c_4 Guests should meet at most once.

Each of these constraints represents a form of template, or schema, with which to instantiate lower-level constraints. All of these lower-level constraints will be in some way related; they are imposing the same type of restrictions but on different variables.

Definition 10 *Given a set of guest boats, G, hosts, H, and periods, P, and a variable matrix, $x_{g,p}$, storing a guest boat's unique host vessel assignment at each period:*

$$\mathcal{C}_3 = \bigcup_{\forall\, g \in G,\, \forall\, p \in P,\, \forall\, q \in P \,:\, p < q} c = \langle S, R \rangle \;\; \textit{where} \; \begin{cases} S = \{x_{g,p}, x_{g,q}\} \\ R = \{x_{g,p} \neq x_{g,q}\} \end{cases} \tag{3.5}$$

The resulting set, $\mathcal{C}_3$, will contain all the disjunction constraints required to restrict the valid assignment of a guest at each period to be unique. However, it would have been equally valid to define c_3 as:

Definition 11

$$\mathcal{C}_3 = \bigcup_{\forall\, g \in G} c = \langle S, R \rangle \;\; \textit{where} \; \begin{cases} S = \{x_{g,p} : \forall\, p \;\in\; P\} \\ R = \{\textit{alldifferent}(S)\} \end{cases} \tag{3.6}$$

Listing 3.1 shows an extract of a COMET model for the PPP. Only the constraints c_2 to c_4 are explicitly modelled. The first constraint—that each guest boat is assigned a host for every period—is satisfied at the initialisation stage. In COMET constraints are instantiated by *posting* them into a container class; most commonly an instance of `ConstraintSystem<LS>`. The contents of that `ConstraintSystem<LS>` are what we would term a family. In subsequent sections we will explore whether this model could have been stated as multiple families (more closely reflecting the problem description).

Neighbourhoods' Behaviours

In the previous section we looked at how constraints could be collected into families. This section investigates how the properties of neighbourhoods can be categorised by the different effects they have on groups of constraints. Previously reasoning about the behaviour of neighbourhoods has been done implicitly. Multi-phase algorithms have their neighbourhoods partitioned based on the understanding that the later neighbourhoods will not introduce violations in the constraints

```
Solver<LS> m();
UniformDistribution distr(Hosts);
var{int} boat[Guests,Periods](m,Hosts) := distr.get();
ConstraintSystem<LS> S(m);
forall(p in Periods){
  S.post(2 * multiknapsack(all(g in Guests) boat[g,p],crew,cap)
    );
}
forall(g in Guests){
  S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
}
forall(i in Guests, j in Guests: j > i){
  S.post(atmost(1,all(p in Periods)(boat[i,p] == boat[j,p])));
}
S.close();
m.close();
```

Listing 3.1: A COMET PPP model excerpted from `ppp-model.co` showing constraints being posted to single `ConstraintSystem<LS>`.

satisfied in the initial phases. At its most basic level a neighbourhood is a function which can generate permutations of assignments. What we have described as the *behaviour* is a product of the interplay between the relation, R_i, of a constraint, c_i, and the projection of all the neighbours.

Definition 12 *If N is a function which maps an assignment (or solution) A to a set of alternate assignments, $N(A)$, then the closure of N, $\mathcal{N}_{closure}(A)$, can be defined as follows:*

$$\begin{aligned}
\mathcal{N}_0 &= \{A\} \\
\mathcal{N}_1 &= N(A) \\
\mathcal{N}_2 &= \bigcup\{N(\bar{a}) : \bar{a} \in \mathcal{N}_1\} \\
\mathcal{N}_i &= \bigcup\{N(\bar{a}) : \bar{a} \in \mathcal{N}_{i-1}\} \\
\mathcal{N}_{closure} &= \bigcup_{i=0}^{\infty} \mathcal{N}_i
\end{aligned}$$

The closure represents all solutions that could be reached from an assignment through applications of N. The corresponding projection for a scope S_i would be $\pi_{S_i}(\mathcal{N}(A)) = \{\pi_{S_i}(\bar{a}) : \forall\, \bar{a} \in \mathcal{N}_{closure}(A)\}$. If this projection is a subset of the constraint's relation, R_i, then any move within N will satisfy c_i. Conversely, if R_i and $\pi_{S_i}(\mathcal{N}(A))$ are completely disjoint then no move within N could ever satisfy c_i. The final alternative is that there is some overlap between the sets,

$|R_i \cap \pi_{S_i}(\mathcal{N}(A))| \geq 1$, meaning a mixture of violating and satisfying assignments is possible. This would determine a neighbourhood's property with regard to a single constraint. Unioning multiple constraints' relations together would allow this to be applied to a family of constraints.

Consider a simple instance of the PPP where there are four guests, $G = \{1, 2, 3, 4\}$, four hosts, $H = \{1, 2, 3, 4\}$ and four periods, $P = \{1, 2, 3, 4\}$. The decision variables representing which host a guest is visiting at each time period will be the same as the one used for Equation 3.5 resulting in sixteen $x_{g,p}$ variables each with the domain $D_{g,p} = H$. If we express c_3 using four **alldifferent** constraints then we would produces the following scopes and relations:

$$\begin{aligned}
S_1 &= \{x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}\} & R_1 = \{&\langle 1,2,3,4\rangle, \langle 1,2,4,3\rangle, \langle 1,3,2,4\rangle, \langle 1,3,4,2\rangle, \langle 1,4,3,2\rangle, \\
& & &\langle 1,4,2,3\rangle, \langle 2,1,3,4\rangle, \langle 2,1,4,3\rangle, \langle 2,3,1,4\rangle, \langle 2,3,4,1\rangle, \\
& & &\langle 2,4,3,1\rangle, \langle 2,4,1,3\rangle, \langle 3,2,1,4\rangle, \langle 3,2,4,1\rangle, \langle 3,1,2,4\rangle, \\
& & &\langle 3,1,4,2\rangle, \langle 3,4,1,2\rangle, \langle 3,4,2,1\rangle, \langle 4,2,3,1\rangle, \langle 4,2,1,3\rangle, \\
& & &\langle 4,3,2,1\rangle, \langle 4,3,1,2\rangle, \langle 4,1,3,2\rangle, \langle 4,1,2,3\rangle\} \\
S_2 &= \{x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4}\} & R_2 &= \ldots \\
S_3 &= \{x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}\} & R_3 &= \ldots \\
S_4 &= \{x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}\} & R_4 &= \ldots
\end{aligned}$$

All the constraints have the same relations but differing scopes. For brevity only R_1 is enumerated; R_2, R_3, and R_4 would be identical. Now let us consider a neighbourhood, N_1, which only operates over the variables in scope S_1. Neighbourhood N_1 is a *2-opt* operator which tries to swap the assignments of the variables in S_1. From the starting projection $\pi_{S_1}(\bar{a}) = \{\langle 1, 2, 3, 4\rangle\}$ the resultant closure of N_1 would be the same as the relation R_1. Since $\pi_{S_1}(N_1(\bar{a})) \subseteq R_1$ all the moves would satisfy the **alldifferent** constraint. If the starting projection had been $\pi_{S_1}(\bar{b}) = \{\langle 1, 2, 1, 4\rangle\}$ then $\pi_{S_1}(N_1(\bar{b}))$ would be:

$$\{\langle 2,1,1,4\rangle, \langle 1,2,1,4\rangle, \langle 1,2,4,1\rangle, \langle 1,1,2,4\rangle, \langle 1,4,1,2\rangle, \langle 1,2,4,1\rangle\}$$

None of the six assignment permutations feature within R_1. To illustrate the third situation—where there is some overlap between the neighbourhood closure and the constraint's relation—we need to introduce a second neighbourhood N_2. This neighbourhood considers assigning new values to the variables in

S_1. Assuming the same starting assignment from the previous example the neighbourhood's projection would be:

$$\{\langle 2,2,1,4\rangle, \langle 3,2,1,4\rangle, \langle 4,2,1,4\rangle, \langle 1,1,1,4\rangle, \langle 1,3,1,4\rangle, \langle 1,4,1,4\rangle$$
$$\langle 1,2,2,4\rangle, \langle 1,2,3,4\rangle, \langle 1,2,4,4\rangle, \langle 1,2,1,1\rangle, \langle 1,2,1,2\rangle, \langle 1,2,1,3\rangle\}$$

The resultant set contains two assignments $\langle x_{1,1}, 3\rangle$ and $\langle x_{1,3}, 3\rangle$ that create the projections $\langle 3,2,1,4\rangle$ and $\langle 1,2,3,4\rangle$ which do feature in the relation R_1. The remaining ten assignments do not feature in R_1, so the choice of assignment determines whether N_2 will satisfy the constraint.

Constraint-Neighbourhood Interactions

In the previous section we saw how the neighbours generated by a neighbourhood may (or may not) appear within the relation of a constraint. This allows us to define what we term a *constraint-neighbourhood interaction.* If a neighbourhood contains a move which can alter the violation state of a constraint, it is said to *interact* with that constraint; Definition 13 states this idea in formal terms:

Definition 13 *For a neighbourhood, N, (generating neighbouring assignments $\bar{a}$), an initial solution, A, and a constraint family, $\mathcal{C}_i$, a constraint-neighbourhood interaction exists iff:*

$$\exists\, \bar{a} \in \mathcal{N}_{closure}(A) \mid violations(\mathcal{C}_i, A) \neq violations(\mathcal{C}_i, \bar{a})$$

In the model from Listing 3.1 all the constraints are being posted into a single instance of the `ConstraintSystem<LS>`; there is effectively only one constraint family. Regardless of neighbourhoods used in any subsequent search component there is no way to differentiate between the violations of constraint families easily. The reason for posting all the constraints to a single `ConstraintSystem<LS>` is primarily historical. In CP all the constraints must be registered with the same Domain Store so that the propagation algorithms can prune the domains of the variables to maintain consistency. The constraints have no communication with each other except via the domains of the variables within their scope. In CBLS no pruning is performed; the constraints provide heuristic guidance. Posting constraints to separate `ConstraintSystem<LS>`s does not reduce any potential information. Modelling differences between CP and CBLS are not unprecedented.

We have already highlighted that Prestwich and Roli [2005] found applying the conventional CP modelling strategy of symmetry breaking was detrimental to the performance of Local Search. Even though all the constraints are posted to a single Domain Store in a CP system, they may still be treated differently internally. Early work, such as Wallace and Freuder [1992], provided guidance on constraint-ordering heuristic strategies. Enforcing arc consistency for some constraints (such as **alldifferent**) is more costly than for other simpler constraints and there is no guarantee that this increased computational investment will result in substantially better pruning. Schulte and Stuckey [2004] cover the benefits of separating constraints into different constraint propagation queues with various levels of prioritisation. So, whilst all the constraints interact via the same interfaces and can have the same domain pruning effects, they may be grouped together based upon some prioritisation decisions made by the designer.

3.2 Modelling Requirements to Represent Constraint Families

The following section investigates whether it is actually feasible to maintain multiple constraint families using COMET. In particular it focuses on whether having multiple constraint families is detrimental to the speed or memory usage of the system. Van Hentenryck and Michel [2007] introduced modelling extensions which increase the level of abstraction provided by COMET to the point that the search and model can be completely decoupled. They provided a new syntax for defining models which encapsulated all the variables and constraints required and allowed these to be passed to generic search components as a single `Model<LS>` object. There are a two main restrictions: constraints can only be tagged using an enumerated value, and the user cannot specify which container type the constraints are stored in. More accurately, the user *can* specify the constraint container type but they *cannot* retrieve constraints other than via `ConstraintSystem<LS>`s. As of version 2.1.1 of COMET the only valid tag values are: `init`, `soft`, `hard`, `min`, `max`, and `minmax`. The generic search procedures provided are: `TabuSearch`, `VNSearch` and a `MinConflictSearch`. Support for these appears tentative and only `TabuSearch` is listed in the documentation.

Listing 3.2 shows the syntax of Van Hentenryck and Michel's **model** and generic search components. The constraints themselves are prefaced with a tag indicating their *type* and also an optional constraint weighting. The tag is used to determine which `ConstraintSystem<LS>` the constraints are posted to. Unlike in Listing 3.1, the constraint posting is hidden from the user and limits the potential families to just two types: hard and soft. For the purposes of our work this scheme is too restrictive as we require the ability to store each constraint family in its own `ConstraintSystem<LS>`. What the system does provide is an example of how it would be possible to allow the definition of families using this neat syntax. There would need to be a way of allowing user-defined enumerated types as tags. Also the "*getter*" methods of the `Model<LS>` class, which at present return only the hard and soft constraints, would require parameterisation so that tagged families could be retrieved.

```
import cotls;
include "genericLocalSearch";

range Guests = 1..4;
range Periods = 1..4;
range Hosts = 1..4;
int cap[Hosts] = [4,3,2,1];
int crew[Guests] = [1,1,1,1];

model m {
  var{int} boat[Guests,Periods](Hosts);
  forall(g in Guests){
    soft(2): alldifferent(all(p in Periods) boat[g,p]);
  }
  forall(p in Periods){
    soft(2): multiknapsack(all(g in Guests) boat[g,p],crew,cap)
    ;
  }
  forall(i in Guests, j in Guests: i < j){
    soft: atmost(1, all(p in Periods)(boat[i,p] == boat[j,p]));
  }
}

TabuSearch search(m);
search.apply();
```

Listing 3.2: A COMET PPP model, `ppp-model-ls.co`, showing the `Model<LS>` syntax and the use of a generic search procedure.

Potential Problems

The additional information gained by maintaining the constraint families as separate `ConstraintSystem<LS>`s must be offset against the increased computational effort. If this technique is to be of practical use, the overhead of multiple `ConstraintSystem<LS>`s should be minimal. To ascertain the cost of this approach we replicated the experiment detailed in Van Hentenryck and Michel [2005, Sec. 10.1.4, pp. 196–204]. In Van Hentenryck and Michel they run their algorithm on the PPP trying 21 different configurations of host vessels and period lengths. The objective of their experiments was to compare the effect of using combinations of intensification and random restart components on the percentage of runs solved and the runtimes and iterations required. As well as these properties, we also monitored the memory usage of the solver to investigate what effect a different constraint posting has. Our experiment used two versions of the PPP algorithm: one that posted all the constraints into a single `ConstraintSystem<LS>` and the other that maintained the constraints as three separate families. Listing 3.3 shows the constraint model being posted to a single `ConstraintSystem<LS>`. The decision about how to post the constraints is described by us as the model's *constraint partitioning*.

```
void setupConstraintSystems(){
  ConstraintSystem<LS> constraintSystem(_solver);

  forall(g in Guests){
    constraintSystem.post(2 * alldifferent(all(p in Periods)(
  boat[g,p])));
  }
  forall(p in Periods){
    constraintSystem.post(2 * knapsack(all(g in Guests)(boat[g
  ,p]),crew,cap));
  }
  forall(i in Guests, j in Guests: j > i){
    constraintSystem.post(atmost(1,all(p in Periods)(boat[i,p]
  == boat[j,p])));
  }
  addConstraint(constraintSystem, "allConstraints");
  populateInvariants();
  _solver.close();
}
```

Listing 3.3: An extract from the single `ConstraintSystem<LS>` version of the COMET model for the PPP defined in the class `ProgressiveParty`, `ppp.co`.

Listing 3.4 shows the entire multiple `ConstraintSystem<LS>` version of the model. The `MultiConstrainedPP` class inherits from the `ProgressiveParty` class. The only method which differs is the `setupConstraintSystems` method that defines how the model is posted; the remainder of the code is identical in both versions. The original model used in Van Hentenryck and Michel [2005] served as the basis for the `ProgressiveParty` model but, to make it more reusable, it was rewritten in an OOP style. The search component remains the same as the one found in Van Hentenryck and Michel's model.

```
class MultiConstrainedPP extends ProgressiveParty{

  MultiConstrainedPP(Solver<LS> m):ProgressiveParty(m){
    // Empty constructor
  }

  void setupConstraintSystems(){

    ConstraintSystem<LS> hostsAllDifferent(_solver);
    forall(g in Guests){
      hostsAllDifferent.post(2 * alldifferent(all(p in Periods)
    (boat[g,p])));
    }
    addConstraint(hostsAllDifferent, "hostsAllDifferent");

    ConstraintSystem<LS> boatCapacity(_solver);
    forall(p in Periods){
      boatCapacity.post(2 * knapsack(all(g in Guests)(boat[g,p])
    ,crew,cap));
    }
    addConstraint(boatCapacity, "boatCapacity");

    ConstraintSystem<LS> guestsMeetAtmostOnce(_solver);
    forall(i in Guests, j in Guests: j > i){
      guestsMeetAtmostOnce.post(atmost(1,all(p in Periods)(boat[
    i,p] == boat[j,p])));
    }
    addConstraint(guestsMeetAtmostOnce, "guestsMeetAtmostOnce");

    populateInvariants();
    _solver.close();
  }
}
```

Listing 3.4: The multiple constraint PPP model, `MultiConstrainedPP`, from `myppp.co`. Only the `setupConstraintSystems` method required overriding.

The experiment consisted of 100 runs for each configuration. Tables 3.1 and 3.2 show the results relating to the search iterations, I, and runtime, T (in ms). There are six different configurations that define which of the boats are designated as hosts. These are listed in Table 3.3. The upper limit for search iterations was fixed at 1,000,000. The results for the memory usage, Tables 3.4 and 3.5, use the same column notation but this time detail the amount of system memory, *Mem*,

Table 3.1: The solution times in iterations I and CPU Time (ms) for the `ProgressiveParty` solver. For a given configuration, C, problem instances can have a range of periods P. The third column, $\%S$, shows the percentage of runs in which the search was able to find a solution that satisfied all the constraints. The columns describe the minimum m, maximum M, mean μ, and standard deviation σ of the runs' iterations I or runtimes T.

C	P	$\%S$	$m(I)$	$M(I)$	$\mu(I)$	$\sigma(I)$	$m(T)$	$M(T)$	$\mu(T)$	$\sigma(T)$
1	6	100	62	100088	1079.21	10000.89	324	64672	986.92	6432.85
	7	100	82	128123	3489.65	18895.38	388	85685	2687.09	12655.49
	8	100	116	100225	13307.83	33748.26	476	71936	9667.84	23486.89
	9	100	164	800379	32075.32	97097.33	604	568807	22957.38	68417.91
	10	99	566	1000000	85586.05	167627.07	944	714012	62027.55	120304.03
2	6	100	76	229	122.80	30.08	336	444	374.40	21.07
	7	100	120	100190	6211.04	23858.93	412	69440	4595.68	16252.74
	8	100	202	200900	13643.72	39279.34	540	136196	9673.28	26609.20
	9	100	741	409313	42877.75	91133.46	1024	287566	30371.69	63682.38
3	6	100	77	283	126.18	27.32	336	484	375.16	21.02
	7	100	107	100220	2224.69	14065.90	396	69164	1846.36	9345.06
	8	100	210	106724	6964.50	24096.30	552	70612	5136.20	16271.85
	9	99	861	1000000	87259.22	142725.79	1108	688018	60633.97	98143.23
4	6	100	94	2279	192.94	222.21	348	1752	420.48	142.82
	7	100	161	100298	1288.12	10001.50	468	68904	1244.52	6834.63
	8	100	247	201222	12199.95	34674.32	596	130928	8543.60	22908.16
	9	100	1268	940418	44169.25	108455.70	1384	644532	30842.52	74268.07
5	6	100	139	111948	4989.34	20294.06	372	72640	3567.40	13270.11
	7	100	486	218013	33716.73	58728.38	704	143296	22688.14	38768.42
6	6	99	177	1000000	76452.62	165741.71	416	657093	50693.87	109356.35
	7	85	1255	1000000	302583.87	355913.05	1200	683686	203300.50	238848.90

in megabytes consumed during the runs. The experimental runs were conducted using COMET version 2.1 on a PC running Kubuntu 8.04 with a 3.4 GHz processor and 2 GB of RAM.

Fig. 3.1 shows the average time spent on each iteration for both partitionings. The curves are similar, although `myPPP`—the multiple constraint version—is skewed to the right. As would be expected the additional computational effort does incur a performance penalty. The difference between the means is only 0.047 ms, so we conclude that using different constraint posting strategies does not markedly increase the average iteration time. This experiment involves trebling the number of `ConstraintSystem<LS>`s but the difference in average iteration times does not reflect this. The iteration time may not have increased; however, it is possible that the memory consumption might have grown unreasonably. The memory usage of each posting scheme is shown in Fig. 3.2. Both plots exhibit the same distinctive bimodal shape and again the `myPPP` results are slightly skewed to

Table 3.2: The solution times in iterations I and CPU Time (ms) for the `MultiConstrainedPP` solver.

C	P	%S	$m(I)$	$M(I)$	$\mu(I)$	$\sigma(I)$	$m(T)$	$M(T)$	$\mu(T)$	$\sigma(T)$
1	6	100	57	100081	1078.77	10000.23	348	69604	1068.40	6922.81
	7	100	80	100134	6129.88	23863.73	432	72412	4730.64	16890.73
	8	100	114	200201	8267.66	30728.51	528	147973	6529.05	22589.01
	9	100	200	200247	13501.38	39330.43	676	148425	10476.62	28989.05
	10	98	469	1000000	95502.65	183052.98	936	766467	72806.16	138529.13
2	6	100	84	100127	2133.38	14069.48	372	69504	1809.28	9693.12
	7	100	112	100182	1215.43	9996.87	468	71896	1261.92	7134.93
	8	100	205	200606	10152.72	32466.08	584	142680	7755.10	23241.32
	9	100	464	401291	38350.17	68293.69	908	294586	28528.89	49942.36
3	6	100	88	252	130.59	28.06	392	524	429.56	21.29
	7	100	130	100192	2230.94	14063.21	484	71884	1943.16	9731.51
	8	100	220	102325	9883.92	28720.55	624	73716	7417.88	20151.45
	9	100	844	804599	68029.14	127952.66	1156	574339	49756.68	91077.16
4	6	100	82	100148	1176.44	9997.37	368	67440	1131.52	6698.04
	7	100	151	100598	4325.60	19704.38	504	71204	3349.96	13423.67
	8	100	246	102918	10304.19	28741.07	652	75328	7766.64	20352.52
	9	100	1129	230853	37909.95	58059.44	1356	166494	27913.75	41860.70
5	6	100	141	100124	1828.39	10100.38	444	68296	1600.04	6856.07
	7	100	277	126130	20848.97	37838.16	596	86953	14897.01	26266.81
6	6	98	250	1000000	85634.12	172884.53	504	686642	59211.44	118696.13
	7	84	459	1000000	318136.93	369284.56	712	711172	223456.45	258634.94

Table 3.3: The Host Boats available for each of the configurations of the PPP.

Config.	1	2	3	4	5	6
Hosts	1–12, 16	1–13	1, 3–13, 19	3–13, 25, 26	1–11, 19, 21	1–9, 16–19

the right. Examining the tables relating to the experiment's memory use shows that the left peak is caused by the configurations which have only six periods. These particular runs consistently use less memory than others regardless of the host configuration or constraint partitioning used. The difference in means for the memory use is 0.356 MB (MB and 29.96 MB); given that this represents only 0.000 17 % of the test machine's total system memory it would seem fair to conclude that storing the constraints in separate `ConstraintSystem<LS>`s does not incur any significant memory overhead. Partitioning models into individual constraint families offers the possibility of more expressive, or informative, heuristics with minimal additional expense.

Table 3.4: The memory usage (in MB) of the `ProgressiveParty` solver.

C	P	$\%S$	$m(Mem)$	$M(Mem)$	$\mu(Mem)$	$\sigma(Mem)$
1	6	100	26.45	29.89	26.50	0.34
	7	100	27.22	37.98	27.49	1.51
	8	100	28.17	38.85	29.60	3.45
	9	100	29.42	39.43	31.60	3.82
	10	99	30.98	39.27	35.24	3.47
2	6	100	26.47	26.59	26.50	0.03
	7	100	27.18	38.41	27.80	2.38
	8	100	28.25	39.16	29.62	3.18
	9	100	29.94	39.50	33.94	3.45
3	6	100	26.46	26.64	26.50	0.02
	7	100	27.18	38.38	27.35	1.15
	8	100	28.27	38.82	29.16	2.41
	9	99	30.05	39.49	36.56	3.15
4	6	100	26.48	27.68	26.56	0.12
	7	100	27.18	36.85	27.31	0.97
	8	100	28.29	38.87	29.57	3.02
	9	100	30.39	39.42	35.18	3.19
5	6	100	26.52	37.55	27.16	1.85
	7	100	27.30	39.10	30.80	3.82
6	6	99	26.55	39.43	29.46	4.10
	7	85	27.95	39.48	35.20	4.25

Compositional Heuristics

When CSP solving with the min-conflicts strategy the heuristic score of a solution is the summation of the violations of all the constraints. Similarly, in SAT the GSAT / WalkSAT algorithms maintain the count of unsatisfied clauses. Each of these algorithms evaluates the fitness of neighbourhood moves based upon their potential to reduce the number of violated constraints / clauses. The presence of plateaux—areas where all the neighbouring solutions return the same score—causes problems for simple improvement algorithms. To the algorithm all the solutions appear equally valid and there is no way of distinguishing them, although the underlying characteristics of the solutions may be quite different. Sutton [2007] investigated the causes of plateaux (or as they term it, landscape neutrality) in scheduling problems. They identify that the one major cause is the range of

Table 3.5: The memory usage (in MB) of the `MultiConstrainedPP` solver.

C	P	$\%S$	$m(Mem)$	$M(Mem)$	$\mu(Mem)$	$\sigma(Mem)$
1	6	100	26.91	37.97	27.10	1.10
	7	100	27.54	38.39	28.12	2.31
	8	100	28.58	39.17	29.36	2.60
	9	100	30.26	39.28	31.46	2.76
	10	98	31.75	39.17	35.60	3.03
2	6	100	26.82	38.10	27.11	1.51
	7	100	27.56	38.48	27.76	1.08
	8	100	28.66	39.22	29.79	2.85
	9	100	30.50	39.46	34.83	3.26
3	6	100	26.82	26.99	26.89	0.04
	7	100	27.58	38.33	27.88	1.48
	8	100	28.67	39.06	29.90	2.81
	9	100	30.85	39.46	36.54	2.92
4	6	100	26.82	38.10	26.98	1.12
	7	100	27.59	38.57	28.08	1.83
	8	100	28.69	39.32	30.16	2.86
	9	100	31.09	39.41	35.97	2.80
5	6	100	26.81	30.08	27.02	0.59
	7	100	27.71	38.83	30.56	3.53
6	6	98	26.81	39.37	30.27	4.28
	7	84	27.87	39.45	35.22	4.20

potential objective values (e.g. the makespan or tardy jobs) may be considerably smaller than the number of neighbouring solutions. Roberts et al. [2005] found that removing neutral moves from the neighbourhood of a satellite scheduling problem harmed performance. Neutrality has also been debated extensively in the Evolutionary Algorithm community. Knowles and Watson [2002] noted that a large fraction of real-life DNA mutations result in neutral changes. This in turn had stimulated researchers to look at artificially adding neutrality into fitness landscapes to potentially aid performance. Ebner et al. [2001] showed neutrality improved diversification by increasing the connectedness of the search space. Knowles and Watson [2002] contradicted this, showing that for evolving Random Boolean Networks (and with a proper mutation rate) redundant mappings actually have no discernible effect on the final solutions' fitness values. For CSPs, many different assignments may result in the same unsatisfied constraints heuristic value.

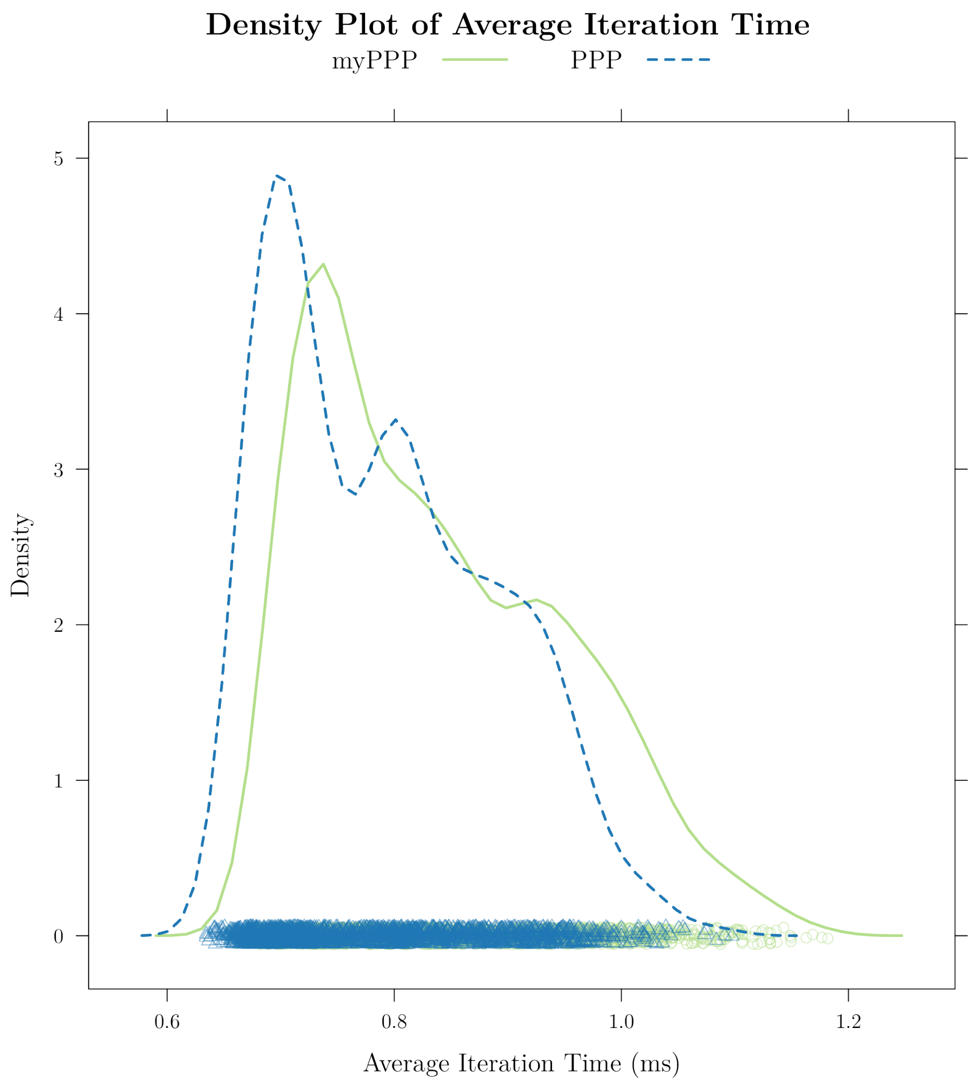


Figure 3.1: The average iteration time (in ms) for each of the constraint partitionings.

We propose that stating a model with multiple `ConstraintSystem<LS>`s gives the potential to disambiguate between these heuristically equivalent solutions.

Number Theory allows us to get an indication of how many violation configurations would result in the same heuristic value. In Number Theory *composition* describes how an integer n could be created by the summation of other positive integers. This does not quite match our situation; it is possible—and indeed desirable—that some of the constraints composing the heuristic will be satisfied (and add zero to the total violations). In this case we need to use what is known as *weak composition* which allows the summation to contain zero values. A valid weak composition of 5 could be $2 + 1 + 1 + 1 + 0$; however, this still does not

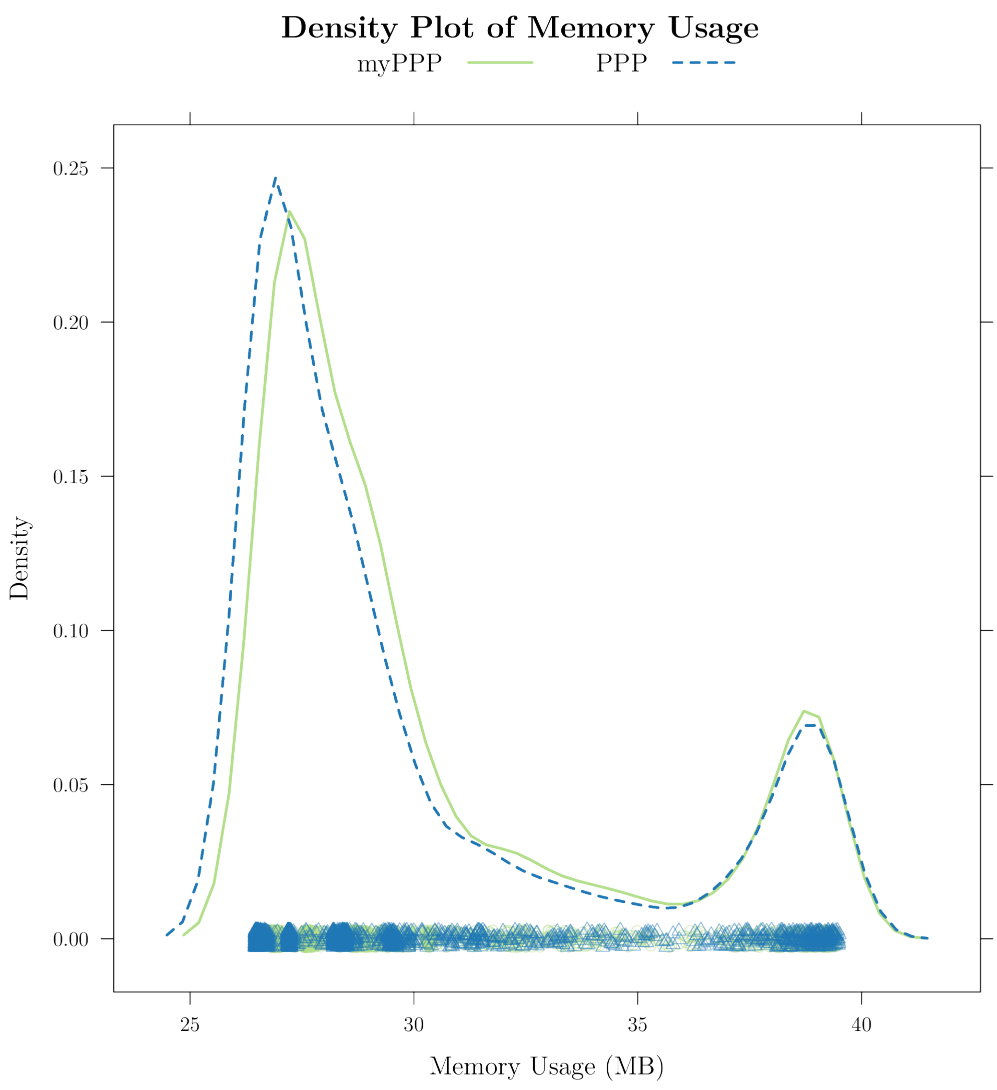


Figure 3.2: The memory usage (in MB) for the different constraint postings.

exactly match our problem because the number of constraint families limits the number of integers the heuristic can be composed from. The answer is to use a tighter form of composition, *weak k-composition*, which restricts the summation to containing exactly k elements. For a given heuristic value, n, and k problem constraint families, then the *weak k-composition*, $C^*_{n,k}$, could be calculated using Equation 3.7. We add the caveat that this value represents an upper bound on the number of different constraint violations which could result in a given heuristic value. Not all configurations may be possible (depending upon the relationships between the constraints). Also, this figure only gives the number of *violation configurations*; there may be multiple assignments which result in the same violations score. If the model contains symmetries there will be at least the number of symmetric solutions for each violation configuration.

$$\mathrm{C}^*_{n,k} = \binom{n+k-1}{k-1} = \frac{(n+k-1)!}{(k-1)!((n+k-1)-(k-1))!} \tag{3.7}$$

It does remain useful though to consider that whilst two solutions may be heuristically identical, this does not mean they are the same and in fact there could be potentially $\mathrm{C}^*_{n,k}$ different violation configurations which would appear identical. By retaining the individual violation information, it is theoretically possible to differentiate between these $\mathrm{C}^*_{n,k}$ solutions.

3.3 Creating a Reusable, Generic Infrastructure for Detecting Constraint-Neighbourhood Interactions

Given that maintaining multiple `ConstraintSystem<LS>`s does not cause any substantial resource overhead and can provide additional information regarding differentiating plateaux neighbours the next question is how to implement a system that allows portable models to be analysed by a problem agnostic Interaction Detector. Most COMET programs described in the literature, or provided as examples with the language, do not make use of the OOP features (such as interfaces or classes). This is, in part, because COMET can succinctly express many problems without them; however, to create a system that allows models to be passed for analysis to a non-problem specific component requires a decoupling of various components. This can be conveniently accomplished using an OOP design. Fig. 3.3 shows a Unified Modelling Language (UML) class diagram highlighting the interfaces and abstract classes which form the core of the Interaction Detector's support framework. The interfaces are coloured light yellow, the abstract classes are green, and the concrete classes are blue. Using COMET as the basis for a higher-level framework is not unprecedented. The *Be-Cool* Belgian Constraints Group at the Université Catholique de Louvain have fielded many applications including: AEON the scheduler solver synthesis system by Monette et al. [2009], the Constraint-Based Graph Matching of le Clément et al. [2009], and `LS(Graph & Tree)` by Dung et al. [2009].

Model

As noted in Section 3.2, COMET's own `Model<LS>` class does not allow users total freedom when posting constraint families. The existing `Model<LS>` interface only permits partitioning models into hard or soft constraints (and objective functions). The constraints are all posted to a `ConstraintSystem<LS>`; there is no way of specifying an alternative constraint container. These limitations motivated the creation of a new `Model` interface which is shown is Listing 3.5. It was designed to allow models to contain an arbitrary number of constraint families and for those families to be stored in any container the user desires. When developing this system we made the decision to focus solely on the integer components of COMET. The language supports integer $\mathbb{Z}$ (**var{int}**), real $\mathbb{R}$ (**var{ float }**), boolean (**var{bool}**) and integer set (**var{set{int}}**) decision variables; the majority of the inbuilt constraints only apply to integer variables. Our framework could easily be extended to cover float, boolean, and set models through the addition of the appropriately-typed methods.

Constraint Container Choice

Allowing constraints to be posted to classes other than just `ConstraintSystem<LS>`s can have efficiency benefits. A specialised form of `ConstraintSystem<LS>` is the `DisequationSystem<LS>` class. It is limited to maintaining only binary constraints of the form: `x[i] != x[j]` (i.e. inequalities). The advantage of using this more limited container is that the differentiation calculations can be performed in constant time [Van Hentenryck and Michel, 2005, p. 111].

To investigate whether the choice of constraint container had an appreciable effect on the model efficiency, we compared two implementations of a Graph K-Colourability problem; one using a `ConstraintSystem<LS>`, the other a `DisequationSystem<LS>`. Our hypothesis was that a `DisequationSystem<LS>` model would be better in terms of differentiation time (i.e. the time it takes COMET to assess the change caused by an assignment). We also wanted to explore whether this speed benefit affected the memory usage and the model's instantiation time. The Graph K-Colourability problem seemed an ideal test problem because the only constraints required are binary inequalities between decision variables that represent the colour of the vertices. The problem—sometimes called the Chromatic Number problem—is known to be $\mathcal{NP}$-hard and appears as [GT4] in Garey and Johnson [1979, p. 191]. The Graph Colouring problem can be defined

```
enum InitialisationMethod = { Random, Permutation, Constructed };

interface Model{

  Solver<LS> getLocalSolver();

  bool hasConstraints();
  ConstraintSystem<LS>[] getConstraints();
  set{int} getConstraintIds();
  set{int} getWeightedConstraintIds();

  bool hasDisequations();
  DisequationSystem<LS>[] getDisequations();
  set{int} getDisequationIds();

  bool hasObjectives();
  Function<LS>[] getObjectives();
  set{int} getObjectivesIds();

  var{int}[] getViolations();
  var{int}[] getEvaluations();

  int[] getSizes();
  string[] getNames();

  var{set{int}} getSelectedConstraints();
  var{int} getSelectedConstraintViolations();

  var{set{int}} getPreservedConstraints();
  var{int} getPreservedConstraintViolations();

  var{set{int}} getSelectedObjectives();
  var{int} getSelectedObjectivesEvaluations();

  bool hasInvariantDependencies();
  dict{int->var{int}[]} getInvariantDependencies();

  void initialiseSolution(InitialisationMethod m);
  void initialiseSolution();

  void populateInvariants();
  Boolean getLookaheadFlag();
  bool isWeighted();

  void print(ostream os);
}
```

Listing 3.5: The `Model` interface, `model.co`, that enables a specific problem to be detached from the other system components.

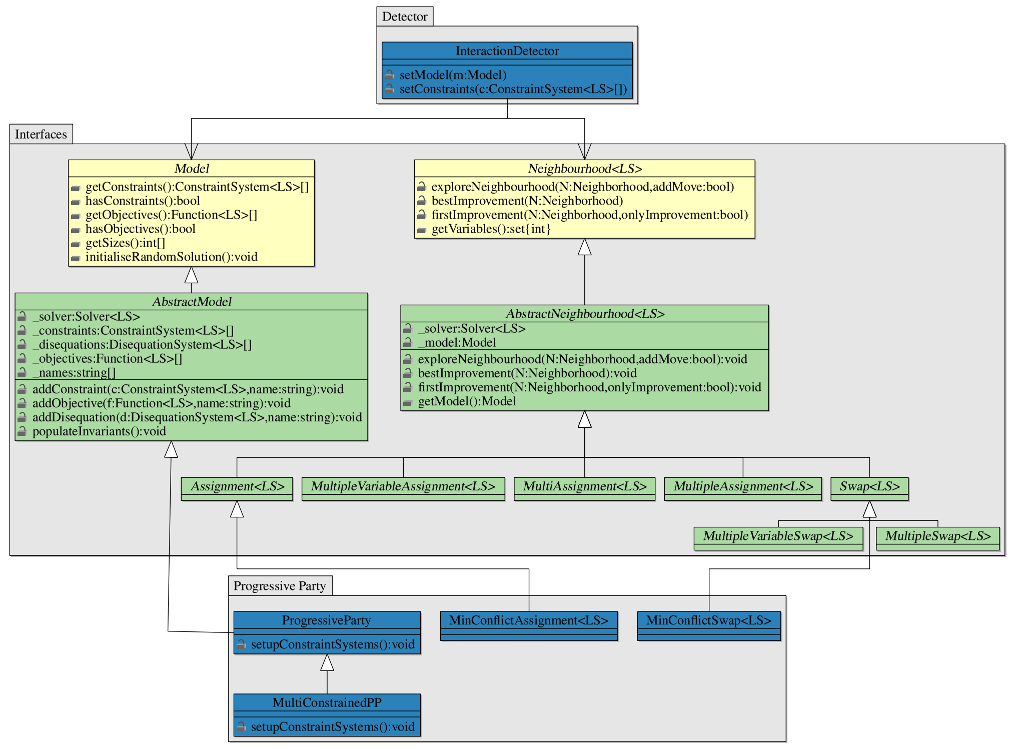

Detector
InteractionDetector
setModel(m:Model)
setConstraints(c:ConstraintSystem<LS>[])
Interfaces
Model
getConstraints():ConstraintSystem<LS>[]
hasConstraints():bool
getObjectives():Function<LS>[]
hasObjectives():bool
getSizes():int[]
initialiseRandomSolution():void
Neighbourhood<LS>
exploreNeighbourhood(N:Neighborhood,addMove:bool)
bestImprovement(N:Neighborhood)
firstImprovement(N:Neighborhood,onlyImprovement:bool)
getVariables():set{int}
AbstractModel
_solver:Solver<LS>
_constraints:ConstraintSystem<LS>[]
_disequations:DisequationSystem<LS>[]
_objectives:Function<LS>[]
_names:string[]
addConstraint(c:ConstraintSystem<LS>,name:string):void
addObjective(f:Function<LS>,name:string):void
addDisequation(d:DisequationSystem<LS>,name:string):void
populateInvariants():void
AbstractNeighbourhood<LS>
_solver:Solver<LS>
_model:Model
exploreNeighbourhood(N:Neighborhood,addMove:bool):void
bestImprovement(N:Neighborhood):void
firstImprovement(N:Neighborhood,onlyImprovement:bool):void
getModel():Model
Assignment<LS>
MultipleVariableAssignment<LS>
MultiAssignment<LS>
MultipleAssignment<LS>
Swap<LS>
MultipleVariableSwap<LS>
MultipleSwap<LS>
Progressive Party
ProgressiveParty
setupConstraintSystems():void
MinConflictAssignment<LS>
MinConflictSwap<LS>
MultiConstrainedPP
setupConstraintSystems():void

Figure 3.3: A UML class diagram showing how the different components and packages of the framework interface with each other.

as:

Definition 14 *Given a graph, G, a $|V(G)|$ sized array of colour assignments, colour, and a set of colours K where $|K| \leq |V(G)|$:*

$$\textit{minimise } |K| \textit{ while } \forall\, e = \{v_1, v_2\} \in E(G) : \textit{colour}[v_1] \neq \textit{colour}[v_2] \tag{3.8}$$

Between 1992 and 1993 The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) organised the second of its *Implementation Challenges* to stimulate research into $\mathcal{NP}$-hard problems (specifically Maximal Clique finding, Graph Colouring, and Satisfiability solving). The input format that DIMACS developed has become the standard for Graph Colouring problems. The instances we used for our experiment were obtained from Michael Trick's repository of DIMACS graph colouring instances[1]. The repository contains a mixture of plain text and compressed binary instances; however, our experiment only used the 57 plain text instances.

The experiment was conducted using COMET 2.1 running on a MacBook Pro operating under Mac OS X 10.6.3 with a 2.16 GHz CPU and 3 GB of RAM. Table B.1 in Appendix B provides the results for every instance. Figs. 3.4 and 3.5 show that memory usage and time to state the constraints is less, in the majority of cases, for a model implemented using the `DisequationSystem<LS>` class. There also appears to be a relationship between the memory usage of a model and the time taken to instantiate that model. Fig. 3.6 shows that the differentiation times were evenly matched. This contradicted our expectations, as in Van Hentenryck and Michel [2005, p. 111] they promote the use of `DisequationSystem<LS>`s precisely due to their constant-time differentiation. Nevertheless, this result highlights that retaining control over the containers used to manage the constraints can have implications for the efficiency of any model.

Differentiation Options

The final argument in favour of posting constraints to containers other than just `ConstraintSystem<LS>`s relates to the differentiation methods available. All constraints (defined over integer variables) inherit from COMET's `Constraint<LS>` interface. This interface specifies five differentiation methods (as shown in Listing 3.6) which can be used to assess the impact of potential assignments on that

[1]The full instance set can be found at: `http://mat.gsia.cmu.edu/COLOR/instances.html`

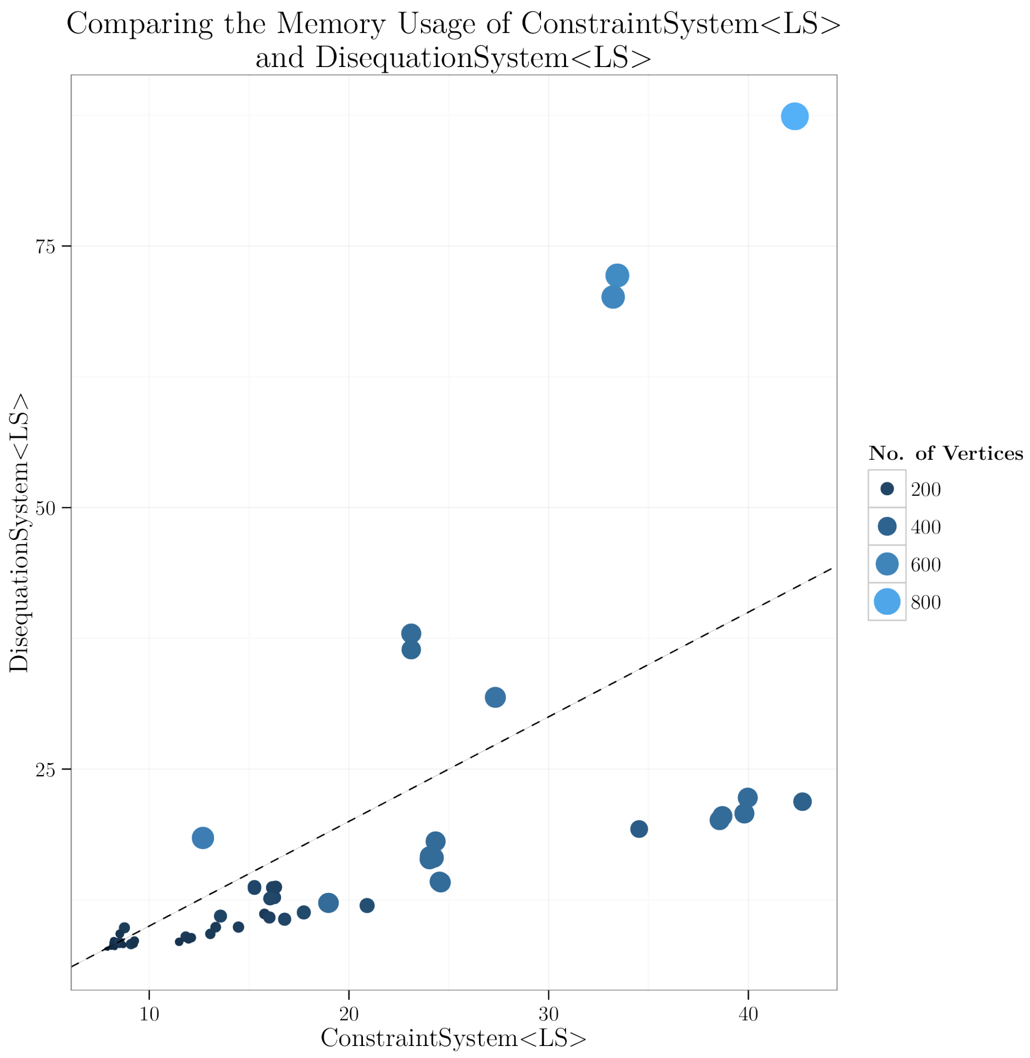


Figure 3.4: Comparing the memory usage (in MB).

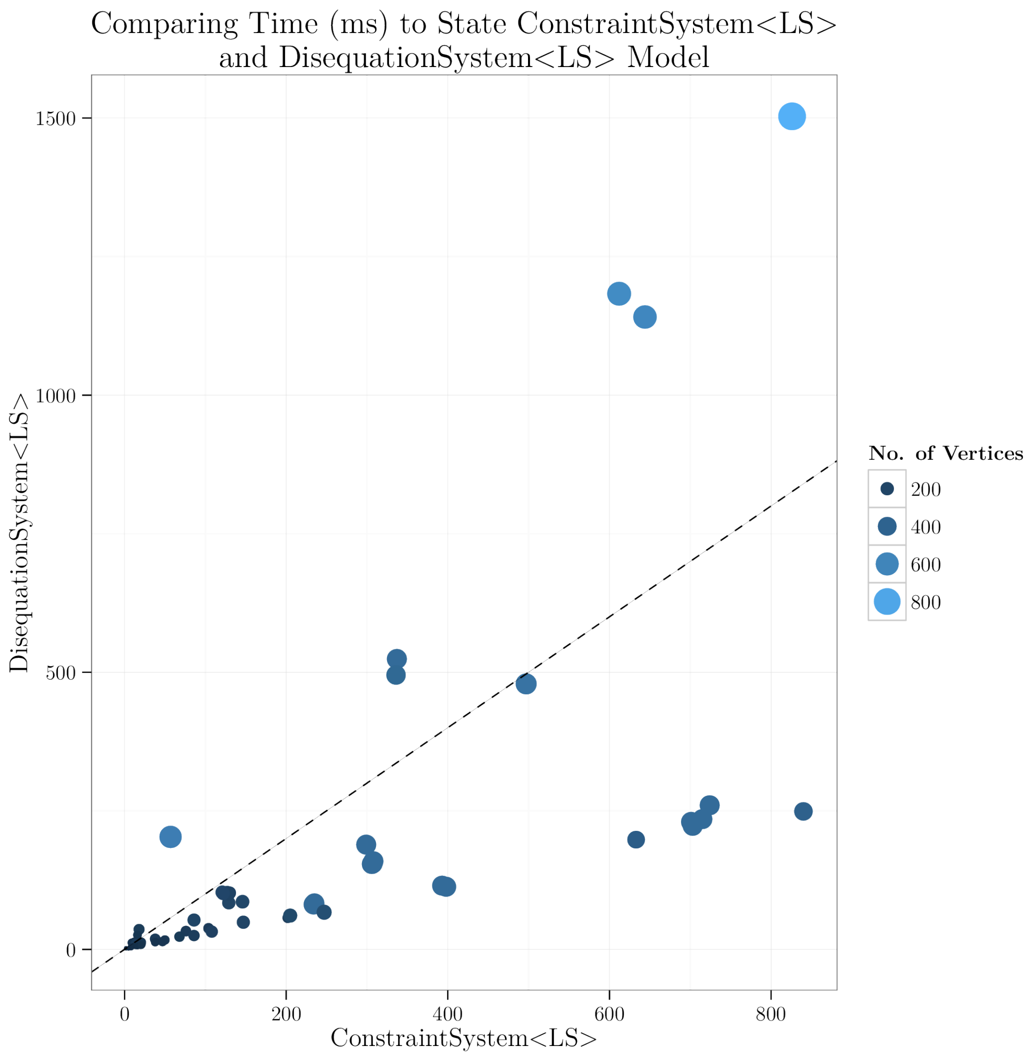


Figure 3.5: Comparing the time (in ms) to state the models.

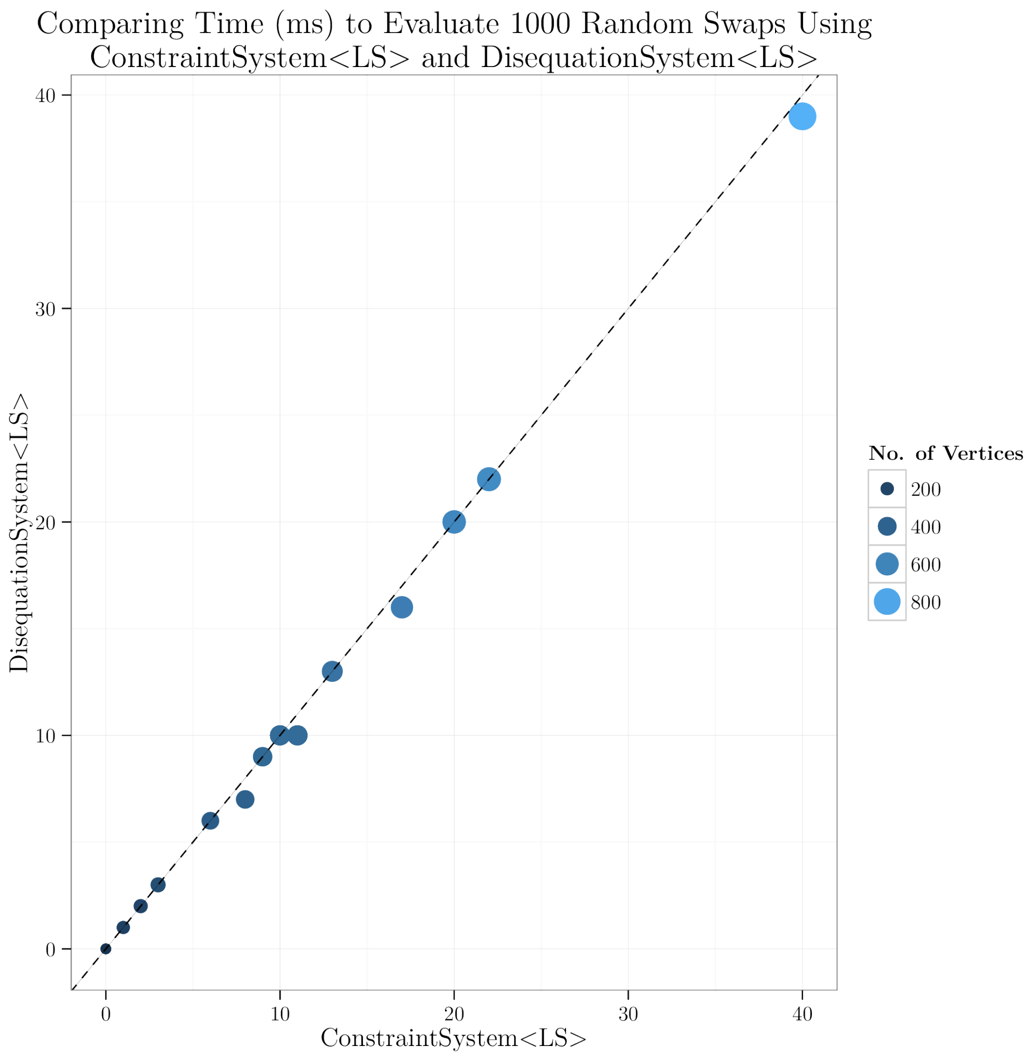


Figure 3.6: Comparing the time (in ms) to differentiate 1000 random swaps.

constraint; these methods are one of the language's major features. Differentiation is a way of efficiently evaluating potential moves without actually committing to them. The `getAssignDelta(var{int} x, int v)` method is the most basic, returning an integer representing the violation change that would have resulted from the call, `x := v`. COMET uses a special operator, `:=`, to make assignments to decision variables. Whenever a decision variable is assigned it causes COMET's underlying machinery to automatically update the appropriate invariants[1]. This process has been made as efficient as possible via the use of incremental algorithms, but it is not without cost. Repeatedly updating invariants is not desirable when searching many neighbours.

```
int getAssignDelta(var{int} x, int v);
int getAssignDelta(var{int}[] xa, int[] va);
int getAssignDelta(var{int} x1, int v1, var{int} x2, int v2);
int getSwapDelta(var{int} x, var{int} y);
int getSwapDelta(var{int} x1, var{int} y1, var{int} x2, var{
  int} y2);
```

Listing 3.6: The differentiation methods provided in COMET's `Constraint<LS>` interface.

The array parameterised version of the `getAssignDelta` method (`getAssignDelta(var{int}[] xa, int[] va)`) is the most powerful differentiation method. It allows the evaluation of moves that reassign multiple variables in a single operation. Without this method it would not be possible to work efficiently with any but the simplest of assignments. The multiple `getAssignDelta` can also be used to recreate the effects of all the other differentiation methods. In addition to assignments, COMET also supports exchanges between decision variables (denoted by the `:=:` operator). Exchanges could be replicated using assignments; however, this would require the creation of a temporary variable to hold the value of one of the variables during the process. The swap operator conveniently solves this problem leading to more concise code.

The `ConstraintSystem<LS>` class implements the `Constraint<LS>` interface thus allowing a collection of constraints to be treated exactly like a single constraint. The downside of this uniformity is that the differentiation possibilities are restricted to *only* these methods. Some constraints provide other, more specialised, differentiation methods; most notably those that implement the

[1]The **atomic** and **delay** blocks can be used to postpone the invariant propagation until the end of several assignments.

SequenceConstraint<LS> interface. In some situations it is more natural to consider potential neighbourhood moves in terms other than assignments or swaps. Problems that depend on precedences or orderings such as Job Shop Scheduling use neighbourhoods that treat the variables as items in a list. Permutations can be created by inserting values at new positions or exchanging the positions of values. As with the swap operation earlier, these permutations are equivalent to a series of assignments, but it is more convenient for the user to have access to a richer set of methods. The SequenceConstraint<LS> interface provides four additional differentiable methods (see Listing 3.7). Posting SequenceConstraint<LS>s into a conventional ConstraintSystem<LS> hides these new methods, instead the SequenceConstraintSystem<LS> should be used. The Model class developed for our framework allows the user control of these decisions.

```
  int getAssignDelta(int x, int v);
  int getReverseDelta(int x, int y);
  int getShiftDelta(int x, int y);
  int getSwapDelta(int x1, int x2);
```

Listing 3.7: The differentiation methods in the SequenceConstraint<LS> interface.

As well as just constraints, COMET also supports objective functions using the same conventions. All *integer* objectives in COMET inherit from the Function<LS> interface that, like its counterpart Constraint<LS>, defines the available differentiation methods. These are limited to single assignments, swaps, and boolean flips (where the truth value of a logical variable is inverted). The Model interface of our framework supports Functions<LS>s in the same ways as constraints. The similarity of the constraint and function interfaces means it requires only minimal additional effort to allow both in our framework.

Implementing our Model interface is an abstract class, AbstractModel, which provides various helper methods to make implementing user-defined problems as simple as possible. The example in Listing 3.4 shows the creation of a model with three ConstraintSystem<LS>s. The modeller can use as many containers as they wish and post their model in any way they see fit. The addConstraint method from the AbstractModel class handles the collation of the individual families into a single array for each of the supported container types.

Invariant Dependencies

When modelling problems, there can be a variety of choices of what to use as the decision variables. In CP it is common to use multiple decision variables to represent the same information from several viewpoints. For example, our model of the Graph K-Colourability problem had a colour value variable for each vertex in the graph; another view could have been to create a set variable for each colour and store the indices of the vertices assigned that colour. The idea of alternate formulations also occurs in Mathematical Programming models where it is described as a *dual*. LP techniques exist which convert between the original—primal—model and the dual to solve problems more effectively. In the case of the dual it refers to a specific recasting of the primal model; alternate viewpoints are not necessarily as restricted in their definitions. In CP and CBLS it can be easier to express certain problem constraints in one particular viewpoint. To ensure that these multiple viewpoints are kept synchronised there needs to be a series of channelling constraints (as detailed in Hnich et al. [2004]).

COMET's Local Search module does not directly support viewpoints or channelling constraints. It is, however, possible to maintain a dual viewpoint by using invariants. As invariants are automatically updated after decision variable assignments channelling constraints are not needed to keep the viewpoints consistent. The limitation is that, unlike in a CP dual model, the relationship only exists in a single direction. It is not valid to manually assign a value to an invariant variable, so bi-directional communication between viewpoints is not possible. Invariant variables appear identical to any other incremental decision variable and can be used within any constraints or functions. When the original decision variables are changed the invariants will update, and so too will the evaluation of any objective or constraint. The problem comes when attempting to differentiate a potential move. The original decision variables are not present in a constraint defined over invariants; any queries based on these will be treated as having no effect. To COMET they are unrelated and it does not recognise there could be any dependency relationship. If the designer is aware of this behaviour then it is possible to use invariants to emulate a dual model. We describe a constraint that is declared on variables other than the decision variables as having an *invariant dependency*.

Our framework is flexible enough to support this non-standard use of the language. When posting the constraints via the `addConstraint` method it is also possible to pass in an additional set of variables to indicate that the constraint depends upon these.

Neighbourhoods

Aside from the model hierarchy, the framework also provides support for easily creating modular neighbourhoods. Neighbourhoods are key to Local Search's effectiveness and as such any Local Search framework should allow the user as much freedom as possible to create suitable functions for their particular problem. The generic search component work in Van Hentenryck and Michel [2007] does not provide any means for the user to specify their own neighbourhoods. The choice is limited to what Van Hentenryck and Michel describe as *linear* and *quadratic* neighbourhoods. The linear neighbourhood equates to straightforward assignments and the quadratic neighbourhood is the exchange of values between variables. As the focus of that work was the development of black-box style Local Search search components, it would be churlish to complain about the omission of user-defined neighbourhoods.

COMET supports neighbourhoods with two main features: the **neighbor** block and the `Neighborhood` interface. The **neighbor** block is a syntactic feature that masks the creation of *closures*. Closures are advanced programming constructs not commonly found in mainstream OOP languages. Essentially, they are first class expressions (i.e. they can be assigned to variables and passed to / from functions) which store a computational state. To execute the code within a closure it needs to be explicitly called. Individual neighbourhood moves can be stored as closures which are only executed once the algorithm has made a decision. The **neighbor** block creates these neighbourhood closures and associates them with a heuristic score (found using the appropriate differentiation method). It then passes the resultant closure and value into a class that implements the `Neighborhood` interface.

The `Neighborhoods` act as closure containers and perform different behaviours regarding which moves they will retain. For example, the `MinNeighborSelector` only stores the closure with the best evaluation. This system of placing moves into a `Neighborhood` makes it easy to compose neighbourhoods by placing moves from a variety of different sources into a single selector. The downside is that it is only the enumerated form of the neighbourhood that is easily transferable. The **neighbor** block must be in the same scope as the decision variables being altered and the differentiable object being used to calculate the fitness.

Our framework attempts to reduce the coupling between the problem, neighbourhood, and search procedure. At the top level is the interface `Neighbourhood<LS>` which defines a number of methods that neighbourhoods must provide. As with the `Model` interface, the majority of these are supplied by an abstract class—`AbstractNeighbourhood<LS>` in this case. In Fig. 3.3 there is a noticeable difference between model and neighbourhood elements; beneath `AbstractNeighbourhood<LS>` are several other abstract classes. These additional classes reflect the various forms of differentiation methods available (as part of the `Constraint<LS>` interface).

From the user perspective, this means that when they are developing their own neighbourhood they need to subclass the appropriate abstract class. If the neighbourhood they intend to create makes multiple assignments (i.e. it would require differentiation with the `getAssignDelta(var{int}[] xa, int[] va)`) then they would extend the `MultiAssignment<LS>` class. Each abstract class provides access to a differentiable method which transparently handles the differentiation of multiple constraint families.

The neighbourhoods support three different search methods: full exploration, first improvement and best improvement. In full exploration mode the neighbourhood creates all the neighbouring solutions and adds them along with their evaluations into the `Neighborhood` for consideration. This is the slowest way to search a neighbourhood and is rarely used within our system. The first improvement exploration only enumerates the neighbourhood until it reaches a point where an improving solution is found. The order of exploration is generated randomly; this should prevent a bias towards moves altering early occurring variables or values. The best improvement method uses COMET's selectors to attempt a min-conflicts style move. The most violated variable is chosen and assigned to the value that causes the largest decrease. As with the first improvement method, the best improvement contains some randomness. COMET selectors can be given a certain *greed* factor. Instead of choosing the best assignment, it can opt for any of the top n options (where n is positive integer that can be varied).

Dynamic Scoping

Another important feature of our neighbourhood system is that the scope of the differentiation functions are not static. In a single `ConstraintSystem<LS>` scenario a move can only be differentiated with respect to that one container.

Our `Model` allows constraints to be posted into multiple containers. When assessing a move, we could call the differentiating method of every container in the `Model`. However, it would be more elegant if moves could be differentiated against only a subset of the available containers (i.e. a dynamic differentiation scope). Each neighbourhood maintains a set containing the indices of the containers that it will include in any differentiation call. These are what we term as the *selected constraints*. The *selected constraints* actually also encompass any disequations and objective functions. Each container in the `Model` has a unique—and contiguous—array index; this is one of the useful functions that happens in the background of the `AbstractModel`. COMET allows arrays to have any range of indices, not just the conventional zero-indexed range. Even though internally the `ConstraintSystem<LS>`s, `DisequationSystem<LS>`s, and `Function<LS>`s are stored as separate arrays to the neighbourhood, they appear as a single array. The default behaviour is to assess the impact of a potential move on all the containers, but the scope can be reduced to any subset of the containers at any point during a search. This offers more flexibility than the existing static differentiation methods.

Lookahead

Even though all the constraints subscribe to the same interface, they may—for various reasons—not support the same differentiation methods. Support for methods beyond the basic `getAssignDelta` and `getSwapDelta` is not guaranteed for all containers and can be buggy; for any classes extending `Function<LS>` it does not exist. Sometimes a constraint has an *invariant dependency* and therefore cannot be differentiated easily. COMET provides a way around this problem in the form of simulation [Van Hentenryck and Michel, 2005, Sec. 8.4, p. 151]. The **lookahead** block allows any arbitrary assignment (or series of assignments) to be executed, returns the effect to a specified incremental variable and then—crucially—rescinds the assignment. By using **lookahead** any unsupported differentiation methods can be simulated; this does incur a larger computational overhead. Listing 3.8 shows a simple example of how **lookahead** could be used to mimic a differentiation method. On line 10 the `ConstraintSystem<LS> S` is being differentiated to find out the effect of assigning variable `x` the value 4. Line 11 shows the alternative way to achieve the same effect using **lookahead**. In this case, it is only the `Solver<LS>` and variable `v`, which stores the violations of `S`, that are supplied. The block following the **lookahead** contains the explicit assignment of

4 to x. The result of the **lookahead** call returns the new value of v, *not* just the change (hence the need for the additional subtraction). Neither method commits the change, so if we examined the value of x after either method it would be 5.

```
import cotls;
Solver<LS> _solver();
var{int} x(_solver, 1..10) := 5;
ConstraintSystem<LS> S(_solver);
S.post(x != 5);
S.close();
var{int} v = S.violations();
_solver.close();

cout << S.getAssignDelta(x, 4) << endl;
cout << lookahead(_solver, v){ x := 4; } - v << endl;
```

Listing 3.8: Comparison of using the **lookahead** construct versus differentiation.

The problem with the existing form of **lookahead** is that it is restricted to returning only a single value. Even though any assignment can be simulated, the effect can only be assessed against a single constraint family. The simplest solution would be to use the same approach as with the differentiation methods and call **lookahead** for each of the containers in the *selected constraints* set. Bearing in mind that each **lookahead** assigns *all* the variables, propagates *all* the invariants, and calculates the values of *all* the constraints, this seems needlessly inefficient. A more effective method would be to only call **lookahead** once and calculate all the changes in a single step. The **lookahead** block does not allow this, tied as it is to a single return value; **lookahead** is a wrapper around a lower-level COMET feature. The Solver<LS> object—which is responsible for maintaining all the variables, invariants, and constraints—contains an undocumented method called lookah(**bool** enable). This can be used in the same fashion as **atomic** and **delay** blocks allowing **lookahead** to occur within a brace delimited scope. The difference between the explicit **lookah** usage and the **lookahead** feature is that primitive assignments between the lookah calls are not undone.

Using **lookahead** raises one more potential performance issue: if the collection of constraint families being differentiated contains a mixture of those which can be differentiated and those which require **lookahead**, then the framework only performs a single **lookahead** operation. Whenever a change is made to the neighbourhood's *selected constraints* set the neighbourhood re-evaluates its **lookahead** policy. This ensures that simulation is only used when it is absolutely necessary.

Candidate Lists

In Glover and Laguna [1997, Chp. 3, p. 61] they outline a feature of TS that they call *candidate lists*. Some neighbourhoods may have too many potential neighbours to make a full exploration feasible. Or, alternatively, the process of evaluating those neighbours could be prohibitively expensive. A candidate list is a way of restricting the number of neighbours to some subset of the full neighbourhood via the application of a set of simple rules. Within our framework, candidate lists are classes which implement an interface and provide a method, `getRestrictedList(int index)`, that can return a set of values for a given parameter. An existing neighbourhood can be customised by supplying a candidate list instance. This makes it straightforward to create multiple neighbourhoods with different properties by supplying different candidate lists to one general neighbourhood.

3.4 The Interaction Detector

In the preceding section we outlined the framework that allows the detector to operate in an entirely problem-independent fashion. This section will describe the system we developed to identify *interactions* between constraint families and search neighbourhoods. Definition 13 specified what constituted a constraint-neighbourhood interaction; a neighbourhood can be said to interact with a constraint family if it contains a move which can alter the violation state of that constraint family. We make no stipulations about whether this change in state is positive or negative. The rationale behind this decision is that, within Local Search, it is the acceptance function's role—not the neighbourhood's—to make choices relating to the heuristic evaluation of states. The neighbourhood exists to provide alternative states for the acceptance function to consider. Given this definition of constraint-neighbourhood interactions, identifying them becomes simple; any observed examples of a violation state change demonstrate that an interaction exists.

The basic structure of the Interaction Detector is shown in Algorithm 10. The algorithm iterates across all the constraint families in the problem model and for each family looks at each neighbourhood. It initialises a random solution as the

Algorithm 10: The structure of the Interaction Detector.

```
input  : An array of $constraints$ and an array of $neighbourhoods$
output : Interaction relationships for each constraint, $cInteractions$
1  begin
2      $C \leftarrow \{1 \ldots |constraints|\}$
3      $N \leftarrow \{1 \ldots |neighbourhoods|\}$
4      foreach $c \in C$ do
5          $cInteractions[c] \leftarrow \emptyset$
6          foreach $n \in N$ do
7              setSelectedConstraint($c, neighbourhood[n]$)
8              try
9                  firstImprovement($neighbourhood[n]$)
10             catch valueChangeEvent(violations($constraints[c]$))
11                 $cInteractions[c] \leftarrow cInteractions[c] \cup \{n\}$
```

starting point and then proceeds to explore the selected neighbourhood. If any of the moves cause a change to the violation state then the relationship is noted and the search investigates the next constraint-neighbourhood pair. If, after exploring all the neighbouring states from the initial random solution, the detectors finds no evidence of a violation change it assumes no interaction exists. The starting solution must exhibit two properties: the constraint under investigation must be violated and the neighbourhood must have a non-zero amount of moves. The second requirement ensures that any neighbourhood is actually allowed to generate neighbours and receives a fair exploration. Some neighbourhoods, especially those which depend upon invariant information, can change in size, from zero moves to many hundreds depending on the current candidate solution configuration.

Due to the flexibility of our framework (especially its treatment of neighbourhoods and constraints as first-class objects), the actual interaction detector algorithm is straightforward. It seems apt to use what is effectively a Local Search to identify properties of another Local Search algorithm. The ParamILS work of Hutter et al. [2009] successfully uses Local Search to tune the parameters of other algorithms, so applying Local Search in this fashion is not unprecedented.

In Equation (3.3) the violations for a constraint family are defined as the number of unsatisfied constraints within that family. Constraints in COMET can report their violations in a number of different forms. For example, an **alldifferent** constraint uses a *variable-based* representation; the violation score reflects how many variables would need to be altered to satisfy the constraint. Constraints can also have a weighting applied to their violations (as appears in the PPP models in Listings 3.1, 3.2, 3.3, and 3.4). Weighting the violations may be a feature of the search strategy (e.g. GLS, SWO) but it can pose problems for the detector. In COMET the violation score maintained by each container incorporates any weightings and so cannot be guaranteed to match Equation (3.3). Under these circumstances a change in violation score may not be proof that the total number of violated constraints has changed. In situations where the model has weighted constraints the detector manually counts the numbers of satisfied constraints. It only identifies an interaction when it detects a *violation* change that also alters the count of currently satisfied constraints.

The complexity of the detector is $O(|C| \cdot |N|)$; where C is the set of constraint families and N is the set of neighbourhoods. The cardinality of C is determined by both the problem specification and the designer's constraint posting decision; we expect this to be somewhere between one and ten. The number of neighbourhoods depends on how many the designer requires (or is willing) to implement. In the literature where Local Search is being used it is unusual to find more than five or six neighbourhoods. Even in papers using VNS the actual number of neighbourhoods is restricted; Burke et al.'s work with VNS uses 23 neighbourhoods which is notably large [2010].

Minor Caveats

Before proceeding to introduce the test domain and the evaluation of the Interaction Detector it would seem prudent to state a few caveats up front. Firstly, and most significantly, failure to detect an interaction does *not* constitute proof that none exists. The detector is *not* complete. Just as Local Search cannot definitively prove SAT instances are unsolvable, it cannot conclusively rule out the existence of a constraint-neighbourhood interaction. The object of this aspect of the work was to see if it was possible to automatically annotate neighbourhoods with information about the constraints they affect. It was never intended to be an exhaustive investigation uncovering all possible interactions.

The second caveat is that the detector makes no attempt to classify the exact nature of an interaction. As mentioned earlier, any change in the number of violations—regardless of the direction—is evidence of an interaction. For most neighbourhoods this will not be a problem, as they will be capable of creating solutions which decrease violations and solutions which increase them. It becomes more of an issue in the situation where a neighbourhood can only ever change the violations in a single direction. It is possible that a neighbourhood contains only either improving or worsening moves (with neutral moves in both); one example is the `overlapRemover` neighbourhood used in the upcoming experiments (in Section 3.6) and summarised in Section A.2.2 of Appendix A. The neighbourhood removes overlapping assignments by setting the undesired allocations to currently empty choices. This neighbourhood will only ever remove overlaps; it cannot introduce new overlaps. This is another reason why the starting solution should contain some violations; some neighbourhoods are able to reduce constraint violations, but if starting from a satisfied position will appear to do nothing. Trying to automatically identify interactions of this nature would be useful but is not possible with the current detection scheme and / or definition of an interaction.

The complexity is $O(|C| \cdot |N|)$, so for larger sets the process can take some time. Especially if there are a lot of combinations where no relationship exists. The final caveat is that the detector is not intended to be used before each solving run. The idea is to use the interaction detector once for a particular problem—independent of a specific *instance*—and set of neighbourhoods; the results uncovered can be cached and loaded later rather than redoing the detection.

To test the effectiveness of the detector we chose to use a variant of the University Timetabling problem. Timetabling problems are challenging combinatorial problems with rich problem constraints where the most successful state-of-the-art solvers are variants of Local Search that implicitly use information about constraint-neighbourhood interactions. It, therefore, seemed that timetabling was an entirely appropriate test domain.

3.5 The Post Enrolment-based Course Timetabling Problem

Timetabling is a classic real-world constraint problem which appears in a variety of settings though is typically situated at educational institutions. The problem is concerned with allocating a series of classes, or events, into locations at predefined time periods. The possible allocations are subject to a series of restrictions; the exact nature of which depends upon the particular timetabling variant. Most of the research has been focused on the problems arising in universities, although other domains include scheduling various types of schools (e.g. Primary, Secondary). The challenging—and repetitive—nature of timetabling means that even from the early days of computing there was research focused on trying to automate the process. There are a number of companies offering commercial timetabling software (such as the UniTime system of Müller [2009] and TimeTabler by Johnson and Johnson [1998]) for use in schools, colleges and universities. Most of these systems operate in a semi-interactive / mixed-initiative fashion; they present a range of potential solutions to the user and allow them to select the most suitable one or reinforce desirable attributes.

The timetabling problem has also been the subject of two international competitions for completely automated timetablers. The particular variant of timetabling used for the first International Timetabling Competition (ITC) is known as *Post Enrolment-based Course Timetabling* (or sometimes Event Timetabling or Course Timetabling). This reflects the situation where students have already been assigned to classes; the challenge is to schedule these classes into time-slots in such a way that students are never required to attend more than one class at any given time. There can only be a single class in each room during a time period and the room must have enough space for the class assigned to it. Any solution which can satisfy these basic requirements is deemed *feasible.* There are a set of additional constraints which indicate desirable attributes in solutions. Students should not be required to attend more than two consecutive classes without a free period. Classes should not be in the final time period of a day and a student should not have any days that contain only a single class.

Definition 15 *A Post Enrolment-based Course Timetabling problem consists of:*

- *a set of m events, $E = \{e_1, e_2, \ldots, e_m\}$*
- *a set of n students, $S = \{s_1, s_2, \ldots, s_n\}$*
- *a set of p rooms, $R = \{r_1, r_2, \ldots, r_p\}$*
- *a set of q room features, $F = \{f_1, f_2, \ldots, f_q\}$*

Also provided in the problem instances are:

- *a list of room capacities, $C = \left\{C_{r_1}, C_{r_2}, \ldots, C_{r_p}\right\}$*
- *an m by n **attends** matrix indicating whether a student takes a particular event.*

- *a p by q* ***roomFeature*** *matrix indicating whether a room has a feature.*
- *a q by m* ***eventFeature*** *matrix storing whether an event needs a feature.*

From the basic information supplied with the problem instance, it is possible to create a more compact, set-representation of some variables. The *attends* matrix can be used to calculate the set of classes that each student is enrolled on:

$$\forall s \in S \ \mathrm{SE}_s = \{\forall e \in E : \mathrm{attends}_{s,e} > 0\}$$

From the studentsEvents (SE) set it is straightforward to create a complimentary structure, *eventsStudents* (EvS), where each event has a set containing the students registered with that class:

$$\forall e \in E \ \mathrm{EvS}_e = \bigcup_{\forall s \in S} \{s : \mathrm{attends}_{s,e} > 0\}$$

The features are supposed to represent particular pieces of equipment either required by a class or present in a room, e.g. an overhead projector, or a white-board. From a practical standpoint, this information can be preprocessed to provide a list of the rooms that meet each event's feature demands. This list can be further reduced by removing any rooms whose capacity is less than the cardinality of the event's student set. The set of all the events that clash with a given event (eC) can be obtained by unioning the events attended by its students.

$$\forall e \in E \ \mathrm{eC}_e \ = \bigcup_{\forall s \in \mathrm{EvS}_e} \mathrm{SE}_s \setminus e$$

The clashing events graph can be augmented with an additional observation: if an event only has one valid room, vR, then it will necessarily clash with any other event which can only be assigned to that same room.

$$\begin{aligned}\forall e \in E \ \mathrm{eC}_e = & \left\{ \bigcup_{\forall s \in \mathrm{EvS}_e} \mathrm{SE}_s \setminus e \right\} \\ & \cup \{f \in E : f \neq e \wedge |\mathrm{vR}_e| = 1 \\ & \wedge |\mathrm{vR}_f| = 1 \\ & \wedge |\mathrm{vR}_e \cap \mathrm{vR}_f| = 1\}\end{aligned}$$

Listing 3.9 contains an excerpt from the `OriginalParser` class which shows how this preprocessing is actually implemented.

```
 studentsEvents = new set{int}[s in students] = setof(e in events)(studentEvents[s,e]);
 eventsFeatures = new set{int}[e in events] = setof(f in features)(eventFeatures[e,f]);
 eventsStudents = new set{int}[e in events] = setof(s in students)(member(e, studentsEvents[s]));

 eventsRooms = new set{int}[e in events] = setof(r in rooms)(and(f in eventsFeatures[e])(roomFeatures[r,f]) && (roomSizes[r] >= card(eventsStudents[e])));
 clashingEvents = new set{int}[e in events] = union(s in eventsStudents[e])(studentsEvents[s]) \ {e}
 union collect(f in events: e != f && card(eventsRooms[e]) == 1 && card(eventsRooms[f]) == 1 && card(eventsRooms[e] inter eventsRooms[f]) == 1)(f);
```

Listing 3.9: An excerpt of the `OriginalParser` class, `OriginalParser.co`, showing the creation of the set-based representation.

Definition 16 *The hard constraints for the ITC [2002] problem are:*

h_1 *no student has to attend more than one event at each time period.*

h_2 *an event's room is large enough to accommodate its students and has the required features.*

h_3 *only one event is in each room at any given time-slot.*

Definition 17 *The soft constraints, which form an objective to be minimised, are:*

s_1 *a student should not have a class in the last time period of a day.*

s_2 *a student should not have more than two events consecutively.*

s_3 *a student should not have only a single event on a day.*

Within the model (and the subsequent experimental results) the constraints have more expressive names. The hard constraints, h_1, h_2, h_3, are referred to as *eventClashes*, *roomValid*, and *overlaps*. Similarly, the soft constraints, s_1, s_2, s_3, are named *finalTimeslot*, *threeInARow*, and *singleEvent*. The *roomValid* constraint is an example of a constraint that has been specified in one form by the competition organisers but could easily be separated into two distinct constraints: an event's room should have enough capacity, and an event's room should have the requested features. In the ITC formulation of the problem there are 45 time-slots; five days, each with nine time periods. This is equivalent to having classes from 9 am until 6 pm, Monday to Friday.

Trying to find to a feasible timetable is closely related to the Graph Colouring problem encountered in Section 3.3. The colours in this case are the time periods, the graph to be coloured is the event clash graph (i.e. any classes sharing students cannot be at the same time). In Graph Colouring the objective is to minimise the number of colours being used; for timetabling it is sufficient to find an assignment which uses, at most, the number of feasible time-slots (i.e. forty).

Timetabling can be neatly modelled as a COP with the problem specification clearly delineating the hard and soft constraints. Listing 3.10 gives an excerpt from the `OriginalModel` class showing the methods responsible for instantiating the constraints *eventClashes* and *overlaps*. Each of the constraint families has its own method where it receives a constraint container as an argument. By making each of the families responsible for posting themselves means that it is easy to create any partitioning desired. For the single partition each constraint family receives the same `ConstraintSystem<LS>`; for multiple families then each family is passed a unique container and so forth.

```
void postClashingEventsConstraints(ConstraintSystem<LS> S){
  forall(e in events, f in clashingEvents[e]:f > e){
    if(_isWeighted){
      S.post(eventTimeslots[e] != eventTimeslots[f], card(
 eventsStudents[e] inter eventsStudents[f]));
    }else{
      S.post(eventTimeslots[e] != eventTimeslots[f]);
    }
  }
}

void postOverlapsConstraints(ConstraintSystem<LS> S){
  S.post(alldifferent(all(e in events) couple(eventTimeslots[
 e], eventRooms[e])));
}
```

Listing 3.10: An excerpt of the `OriginalModel` class (from `OriginalModel.co`) showing the *eventClashes* and *overlaps* constraints being posted.

Experimental Problem Model

The problem model we decided to use has a decision variable for each event's room and time-slot assignment. An alternative approach is to a have a decision variable for each time-slot room combination (where the domain would be the range of events). The benefit of the second approach is that it prevents *overlaps* implicitly; each time-slot room variable can only take on a single event assignment. The downside is that it makes modelling the *threeInARow* constraint more demanding. Dynadec provide a demonstration timetabling application as one of their COMET examples in Dyn [2010, Sec. 19.4, p. 373][1]. This model uses a time-slot / room formulation and takes its input in the ITC format but does not fully capture the ITC problem. Several of the constraints (*finalTimeslot*, *singleEvent*, and *threeInARow*) are omitted. The *finalTimeslot* constraint would be fairly trivial to encode in this model; just remove all the events from the domains of any of the final time-slot / room variables.

The remaining constraints—*threeInARow* and *singleEvent*—are the hardest to model; both are violated on a per student basis. The most elegant way to capture the *threeInARow* constraint is using the `SequenceAtmost<LS>`. The `SequenceAtmost<LS>` is COMET's Local Search version of the CP `sequence` global constraint introduced by Dincbas et al. [1988] to model car production-line problems. This requires a representation that maintains each student's timetable explicitly. To use the `SequenceAtmost<LS>`, we need access to each student's daily timetable (or a count of the events they have at each time-slots during that day). Having a decision variable for every time-slot in each student's timetable is not feasible; the search process would become unworkable. However, automatically maintaining each student's timetable as invariant *is* possible. COMET allows users to extend the `Invariant<LS>` interface to efficiently update any custom invariants they want. Ordinarily, invariants should not be used in constraints, however, in the case of the *threeInARow* (and *singleEvent*) constraints it seemed the neatest option (and motivated our desire to handle *invariant dependencies* in the framework).

[1]The source file is `timetableLS.co` which can be found in the `docs/codes/cbls` folder which is located within the COMET installation directory.

Additional Instances

The ITC provided twenty instances created by Ben Paechter which were guaranteed to have *perfect solutions*, i.e. where all the hard and soft constraints could be satisfied. In addition to the public instances the organisers also held three private instances in reserve that they could use to independently verify the performance of submissions. After the competition Lewis (using Ben Paechter's problem generator) released another sixty instances[1] designed to be *harder* than the original competition ones [2006, Sec. 4.3, p. 59]. Not all of these instances are known to be soluble to the lower bound; they may not have perfect solutions.

Previous Approaches

The original competition resulted in a number of approaches; the overall winner Kostuch [2004] used SA hybridised with constructive components. The second placed entry of Cordeau et al. [2003] used a TS with several neighbourhoods. Third placed Bykov [2003] expanded upon the earlier work of Burke et al. [2001] and used the Great Deluge Algorithm (GDA). Fourth place was awarded to Di Gaspero and Schaerf [2003a,b, 2006] who used their EasyLocal++ framework to create a VNS based algorithm. As part of the rules of the competition each entrant had to supply a short technical report detailing their algorithm; the papers for Bykov [2003], Chiarandini et al. [2003], Cordeau et al. [2003], Di Gaspero and Schaerf [2003b], Kostuch [2004] can be obtained from the ITC results page at `http://www.idsia.ch/Files/ttcomp2002/results.htm`.

The most effective algorithm was actually a submission by the competition organisers (and thus ineligible to win). For details, check their original technical report in Chiarandini et al. [2003] or the later, more expansive article in Chiarandini et al. [2006]. Their approach used a combination of different metaheuristic components. The first phase used a constructive algorithm coupled with a TS to find a feasible solution. Once a feasible solution was found, a VNS was executed trying to optimise the soft constraints; this was passed to a final SA phase to try to reduce any remaining constraint violations.

All the successful entries used some kind of metaheuristic algorithm. It may be just because the ITC was organised by the Metaheuristics Network, but it is interesting that no CP or LP solutions featured. Whilst the specific approaches

[1]The *harder* instances are available from `http://www.dcs.napier.ac.uk/~benp/centre/timetabling/harderinstances.htm`

varied between submissions, a common theme between them all was taking a multi-phase approach and partitioning the problem into the hard and soft constraints. The best algorithms actually used several phases and were non-linear in their construction (e.g. they could move between several phases in a non-trivial fashion).

Work presented in Lü et al. [2010] also uses the timetabling problem—albeit the Curriculum-based rather than Post-enrolment based variant—as the setting for neighbourhood analysis. Their work investigates four neighbourhoods and focuses on three properties unrelated to the constraints: the strength of moves, the percentage of improving neighbours and the number of search steps. The neighbourhoods they use are *SimpleMove*, *SimpleSwap*, *KempeMove*, and *KempeSwap*. The first three are equivalent to `moveToEmptySlot`, `eventsTimeslotAndRoomSwaps`, and `kempeChainWithMatching` in our framework. We do not have a neighbourhood with the same behaviour as their *KempeSwap*.

3.6 Evaluating the Interaction Detector

To test the effectiveness of the Interaction Detector we experimented using the timetabling problem with 41 neighbourhoods and three different constraint partitionings. Table 3.6 summarises the neighbourhoods and their properties; a full description of each neighbourhood can be found in Appendix A. To try and keep the strength and size formulas concise we use the following abbreviations. Definition 15 covers the abbreviations that are properties of the problem instances.

D the set of days in a week.

tPD the number of time-slots per day (i.e. 9).

Finals the set of time-slots that are the last on each day.

fE the set of events that are assigned to time-slot in $Finals$.

ES the empty slots, those which contain no event assignments.

$\mathbf{eSOD}_d$ the empty slots on a particular day d.

$\mathbf{eOD}_d$ the set of events on a day d.

$\mathbf{ER}_r$ the set of events in room r.

$\mathbf{E}_{t_i}$ the set of events at time-slot t_i.

OE the set of overlapping events, i.e. those in the same time-slot and room.

cl the chain length for an Ejection Chain neighbourhood.

vR the set of valid rooms that satisfy an event's size and feature requirements.

$\mathbf{eC}_e$ all the events that clash with an event, e; i.e. those with which it either shares students or has the same single valid room.

$\mathbf{C}_e$ the current clashes for an event e; i.e. a subset of eC_e that are assigned the same time-slot as e.

Listing A.1 in Appendix A contains an excerpt from the `OriginalModel` class where many of these invariants are declared.

We manually identified the interactions of every constraint-neighbourhood pair and used that as an answer scheme against which the detector's effectiveness could be compared. The first partitioning has all of the constraints posted into a single container; this is in line with a traditional CP style model. The second partitioning separates the hard and soft constraints into their own containers; this is using the COP style of partitioning. The final partitioning gives each of the six problem constraint families a separate container. We used four different problem instances: `competition01.tim` and `competition02.tim` from the ITC set; `small_1.tim` and `small_2.tim` from the *harder* set. The experimental results were gathered by distributing the runs across a laboratory of 30 PCs each running 64 bit Ubuntu 10.04 with a dual core 3.2 GHz processor and 8.0 GB of RAM. COMET version 2.1.1 was used. The Interaction Detector has the ability to display a visualisation of the detection process occurring; a screenshot can be found in Fig. 3.12. During the experimental runs the GUI was not enabled.

Results Charts

Figs. 3.7, 3.8, and 3.9 show the results of the detector on the three different model partitionings. The figures are designed to emulate the *Consumer Reports* car frequency-of-repair comparison chart found in Tufte [2001, p. 174]. They can be thought of as a type of Hinton Diagram [1986]. Each chart shows a matrix of neighbourhoods and constraints. The green circles represent relationships that *should* be detected. If the detector uncovers a relationship where none exists

Table 3.6: Summary of the 41 neighbourhoods implemented.

	Name	Move Type	Ma Strength	Max Size	Generic	Candidate List
1	`allEventsTimeslotSwaps`	`SwapAllEventsTimeslots<LS>`	$2 \cdot \lvert R\rvert$	$\binom{\lvert T\rvert}{2}$	No	No
2	`allEventsTimeslotSwapsInDay`	`SingleDaySwapAllEventsTimeslots<LS>`	$2 \cdot \lvert R\rvert$	$\lvert D\rvert \cdot \binom{tPD}{2}$	No	No
3	`allEventsTimeslotSwapsInDayMinusFinals`	`SwapAllEventsTimeslots<LS>`	$2 \cdot \lvert R\rvert$	$\lvert D\rvert \cdot \binom{tPD-1}{2}$	No	Yes
4	`allEventsTimeslotSwapsMinusFinals`	`SwapAllEventsTimeslots<LS>`	$2 \cdot \lvert R\rvert$	$\binom{\lvert T \setminus Finals\rvert}{2}$	No	Yes
5	`clashDirectedAssignmentsMinusFinals`	`ClashDirectedAssignment<LS>`	1	$\lvert\{\forall e \in E : \lvert C_e\rvert > 0\}\rvert \cdot \lvert T \setminus Finals\rvert - 1$	No	Yes
6	`consistentClashDirectedSwaps`	`ClashDirectedAllSwaps<LS>`	2	$\leq \lvert E\rvert \cdot (\lvert T\rvert - 1)$	No	No
7	`consistentClashDirectedSwapsInDay`	`ClashDirectedAllSwaps<LS>`	2	$\leq \sum_{d=1}^{\lvert D\rvert} \lvert eOD_d\rvert \cdot (tPD - 1)$	No	Yes
8	`consistentClashDirectedSwapsInterDay`	`ClashDirectedAllSwaps<LS>`	2	$\leq \lvert E\rvert \cdot ((\lvert D\rvert - 1) \cdot tPD)$	No	Yes
9	`consistentInterDaySwaps`	`ConsistentInterDayAllSwaps<LS>`	2	$\binom{\lvert E\rvert}{2} - \sum_{d=1}^{\lvert D\rvert} \binom{\lvert eOD_d\rvert}{2}$	No	No
10	`consistentRoomSwaps`	`ConsistentAllSwaps<LS>`	2	$\lvert E\rvert \cdot \binom{\lvert R\rvert}{2}$	Yes	No
11	`consistentTimeslotSwaps`	`ConsistentAllSwaps<LS>`	2	$\sum_{r=1}^{\lvert R\rvert} \binom{\lvert ER_r\rvert - O_r}{2}$	Yes	No
12	`consistentSingleDaySwaps`	`ConsistentSingleDayAllSwaps<LS>`	2	$\sum_{d=1}^{\lvert D\rvert} \sum_{r=1}^{\lvert R\rvert} \binom{\lvert ER_r \cap eOD_d\rvert - \lvert O_{d,r}\rvert}{2}$	No	No
13	`ejectionChain`	`EjectionChain<LS>`	$2 \cdot cl$	$\leq \lvert E\rvert \cdot (\lvert R\rvert\lvert T\rvert - (1 + \lvert ES\rvert))^{cl} \cdot \lvert ES\rvert$	No	No
14	`ejectionChainInDay`	`EjectionChain<LS>`	$2 \cdot cl$	$\leq \lvert E\rvert \cdot (\lvert R\rvert\lvert T\rvert)^{cl} \cdot \lvert ES\rvert$	No	Yes
15	`ejectionChainInDayMinusFinals`	`EjectionChain<LS>`	$2 \cdot cl$	$\leq \lvert E\rvert \cdot (\lvert R\rvert\lvert T\rvert)^{cl} \cdot \lvert ES\rvert$	No	Yes
16	`ejectionChainMinusFinals`	`EjectionChain<LS>`	$2 \cdot cl$	$\leq \lvert E\rvert \cdot (\lvert R\rvert\lvert T\rvert)^{cl} \cdot \lvert ES\rvert$	No	Yes
17	`eventsTimeslotAndRoomSwaps`	`AllMultiVariableSwaps<LS>`	4	$\binom{\lvert E\rvert}{2}$	Yes	No
18	`kempeChain`	`KempeChain<LS>`	$\lvert E_{t_1} \cup E_{t_2}\rvert$	$\leq \binom{\lvert T\rvert}{2} \cdot 2 \cdot \lvert R\rvert$	No	No
19	`kempeChainInDay`	`KempeChain<LS>`	$\lvert E_{t_1} \cup E_{t_2}\rvert$	$\leq \sum_{d=1}^{\lvert D\rvert} \binom{tPD}{2} \cdot 2 \cdot 2 \cdot \lvert R\rvert$	No	Yes
20	`kempeChainInDayMatching`	`KempeChain<LS>`	$4 \cdot \lvert R\rvert$	$\binom{tPD}{2}$	No	Yes
21	`kempeChainInDayMinusFinals`	`KempeChain<LS>`	$\lvert E_{t_1} \cup E_{t_2}\rvert$	$\leq \sum_{d=1}^{\lvert D\rvert} \binom{tPD-1}{2} \cdot 2 \cdot \lvert R\rvert$	No	Yes
22	`kempeChainInDayMinusFinalsMatching`	`KempeChain<LS>`	$4 \cdot \lvert R\rvert$	$\leq \sum_{d=1}^{\lvert D\rvert} \binom{tPD-1}{2} \cdot 2 \cdot \lvert R\rvert$	No	Yes
23	`kempeChainMinusFinals`	`KempeChain<LS>`	$\lvert E_{t_1} \cup E_{t_2}\rvert$	$\leq \binom{\lvert T \setminus Finals\rvert}{2} \cdot 2 \cdot \lvert R\rvert$	No	Yes
24	`kempeChainMinusFinalsMatching`	`KempeChain<LS>`	$4 \cdot \lvert R\rvert$	$\leq \binom{\lvert T \setminus Finals\rvert}{2} \cdot 2 \cdot \lvert R\rvert$	No	Yes
25	`kempeChainWithMatching`	`KempeChain<LS>`	$4 \cdot \lvert R\rvert$	$\leq \binom{\lvert T\rvert}{2} \cdot 2 \cdot \lvert R\rvert$	No	No
26	`moveTimeslotAndRoom`	`DomainMultipleVariableAssignment<LS>`	2	$\lvert R\rvert \cdot \lvert T\rvert - 1$	Yes	No
27	`moveTimeslotAndRoomMinusFinals`	`DomainMultipleVariableAssignment<LS>`	2	$\lvert R\rvert \cdot \lvert T \setminus Finals\rvert - 1$	Yes	Yes
28	`moveToEmptySlot`	`MoveToEmptySlot<LS>`	2	$\lvert E\rvert \cdot \lvert ES\rvert$	No	No
29	`moveToEmptySlotInDay`	`MoveToEmptySlot<LS>`	2	$\sum_{d=1}^{\lvert D\rvert} \lvert eOD_d\rvert \cdot \lvert eSOD_d\rvert$	No	Yes
30	`moveToEmptySlotInDayMinusFinals`	`MoveToEmptySlot<LS>`	2	$\sum_{d=1}^{\lvert D\rvert} \lvert eOD_d\rvert \cdot \lvert eSOD_d\rvert$	No	Yes
31	`moveToEmptySlotMinusFinals`	`MoveToEmptySlot<LS>`	2	$\sum_{d=1}^{\lvert D\rvert} \lvert eOD_d \setminus fE\rvert \cdot \lvert eSOD_d \setminus Finals\rvert$	No	Yes
32	`overlapRemover`	`OverlapRemovingAssignment<LS>`	2	$\lvert OE\rvert \cdot \lvert ES\rvert$	No	No
33	`roomAssignments`	`SetAssignment<LS>`	1	$\lvert E\rvert \cdot (\lvert R\rvert - 1)$	Yes	No
34	`roomMatching`	`RoomMatching<LS>`	$\lvert R\rvert$	$\lvert T\rvert$	No	No
35	`singleDayAllEventsTimeslotSwaps`	`SwapAllEventsTimeslots<LS>`	$2 \cdot \lvert R\rvert$	$\lvert D\rvert \cdot \binom{tPD}{2}$	No	Yes
36	`singleDaySwaps`	`SingleDayAllSwaps<LS>`	2	$\sum_{d=1}^{\lvert D\rvert} \binom{\lvert eOD_d\rvert}{2}$	No	No
37	`timeslotAssignments`	`DomainAssignment<LS>`	1	$\lvert E\rvert \cdot (\lvert T\rvert - 1)$	Yes	No
38	`timeslotAssignmentsMinusFinals`	`SetAssignment<LS>`	1	$\lvert E\rvert \cdot \lvert T \setminus Finals\rvert - 1$	Yes	Yes
39	`validRoomAndTimeslotSwaps`	`ValidRoomAndTimeslotSwaps<LS>`	4	$\binom{\lvert E\rvert}{2}$	No	No
40	`validRoomAndTimeslotSwapsInDay`	`ValidRoomAndTimeslotSwaps<LS>`	4	$\sum_{d=1}^{\lvert D\rvert} \binom{\lvert eOD_d\rvert}{2}$	No	Yes
41	`validRoomAssignments`	`DomainAssignment<LS>`	1	$\sum_{e=1}^{\lvert E\rvert} \lvert vR_e\rvert - 1$	Yes	No

(analogous to a statistical type I error, i.e. a *false positive*) then it would be displayed in the matrix as a red triangle. The amount of fill within the shapes represents the proportion of the test runs which generated that result; for shapes that are not completely filled the exact ratio is displayed as a fraction to the right. If the detector is 100 % accurate the results matrix would show only filled green circles. The fill of the circles is scaled so that in the event of an interaction never being detected the chart would still show a circle with a thin, green outline. We call the set of constraints that a neighbourhood interacts with its *signature*.

Single Partition Results

The results (in Fig. 3.7) where all the constraints are grouped into a single `ConstraintSystem<LS>` show that all the neighbourhoods have the same number of interactions. This should perhaps come as no surprise. If a neighbourhood did not interact with at least one of the constraints in the composite `allConstraints` system then it would not be useful for solving timetabling problems.

Fig. 3.8 shows the results from partitioning the constraints into the families covered in Definitions 16 and 17. Differences between the neighbourhoods can now start to be seen. From the two-phase partition there could be at most four unique interaction signatures. This is because the interaction property is binary; either there is an interaction or not. With two partitions there are two binary variables giving 2^2 potential outcomes. The first neighbourhood, `allEventsTimeslotSwaps`, only interacts with *softConstraints*. There are four neighbourhoods (`roomAssignments`, `roomMatching`, `consistentRoomSwaps`, and `validRoomAssignments`) that are limited to just the hard constraints. Five of the neighbourhoods only interact with the soft constraints. As with the single partition results, the detector is able to find all the expected interactions in every run and does not misclassify any interactions.

Full Partition

For the final partition scheme the results in Fig. 3.9 uncover more differences between the neighbourhoods. These results give the finest granularity and allow disambiguation between neighbourhoods which still appeared identical under the two-phase partition. The additional constraint families means there could be a maximum of 2^6 (i.e. 64) distinct signatures; only 26 actually appear. Out of the 41 neighbourhoods there are only 9 that affect only a single

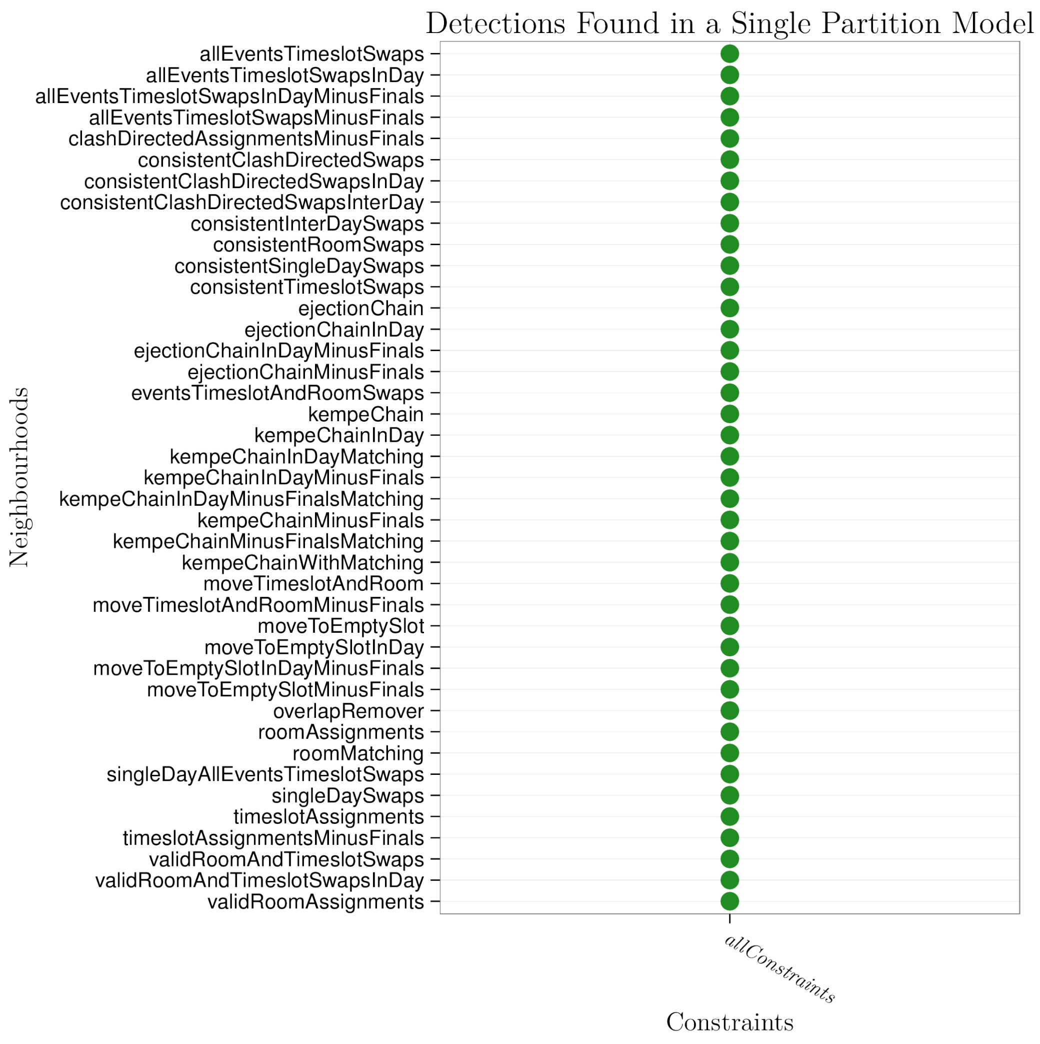


Figure 3.7: Interaction detections found in a single partition model.

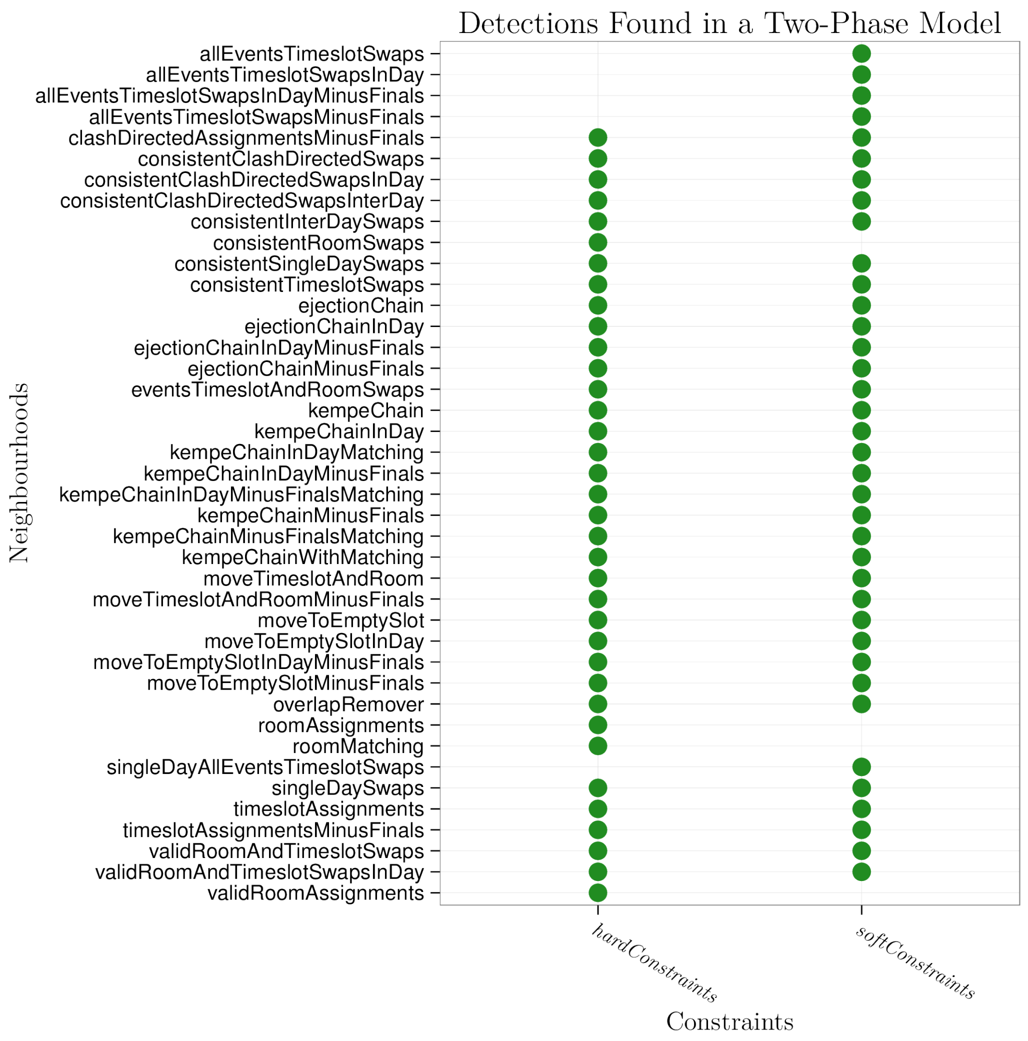


Figure 3.8: Interaction detections found in a two-phase model.

of the constraint families in the two-phase partition. Moving to the full partition model shows that the other 32 neighbourhoods can be separated further; amongst the previously identical neighbourhoods there emerge 19 distinct signatures. Ten signatures are unique, i.e. they appear for only one neighbourhood. The remaining 22 neighbourhoods are distributed across 9 signatures. The maximum number of indistinguishable neighbourhoods is now four; `ejectionChain`, `moveTimeslotAndRoom`, `moveToEmptySlot`, and `overlapRemover` affect all six constraint families. The quartet of neighbourhoods `consistentClashDirectedSwaps`, `consistentClashedDirectedSwapsInterDay`, `consistentInterDaySwaps`, and `consistentTimeslotSwaps` all have the signature *eventClashes*, *threeInARow*, and *singleEvent*. The detector's accuracy remains reliable; it makes no misclassifications, and only `kempeChainInDayMatching` and `kempeChainInDayMinusFinalsMatching` fail to uncover their relationship with the *finalTimeslot* constraint in all cases. Both of these neighbourhoods are compound neighbourhoods that really represent two distinct moves: the Kempe Chain itself and a secondary graph matching operation. If the second component fails to find a valid matching no move is returned. Depending on the solution that it starts from, the matching component may not find a feasible room assignment. In this case it will not return any candidate solutions to the detector. This behaviour mirrors the $\mathcal{N}_4$ neighbourhood in Chiarandini et al. [2003] where they intentionally discarded infeasible matchings to ensure the neighbourhood would preserve the *hardConstraints*. The result for the detector is that if there are no moves to execute, it cannot find interactions, because the detector relies on the events being thrown when a solution is assigned. This `kempeChain` / matching hybridisation is unusual but does not prevent the detector from correctly identifying its relationships in the majority of cases.

A Visual Comparison of Interaction Signatures

Whilst Figs. 3.7, 3.8, and 3.9 provide an overview of all the interactions for a particular partitioning, they make it more challenging to chart the way that a neighbourhood's interaction signature changes across the three partitionings. Figs. 3.10 & 3.11 provide an alternative viewpoint of the data by only focusing on the relationships discovered for two neighbourhoods: `moveTimeslotAndRoom` and `consistentSingleDaySwaps`. To make for a more concise diagram the universal quantifier symbol, $\forall$, is used to represent *allConstraints*, H for *hardConstraints*, and S for *softConstraints*. The remaining node names correspond to those

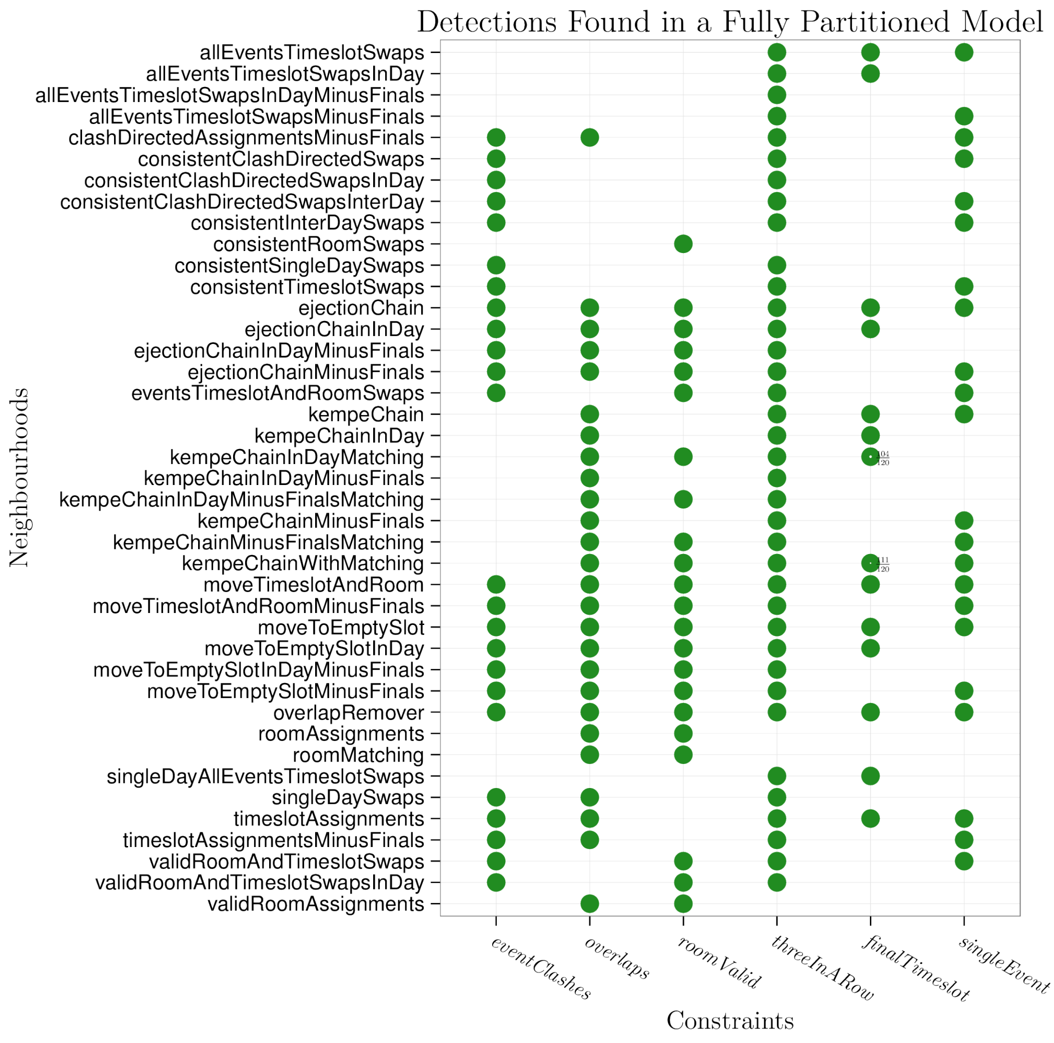


Figure 3.9: Interaction detections found in a fully partitioned model.

found in Definitions 16 & 17. Each layer in the tree is a constraint partitioning, and a node's truth value should be the conjunction of its children. The edges indicate where relationships exist; a disconnected node has not been uncovered as an interaction. Both neighbourhoods appear identical in the first two layers (i.e. under the single and two-phase partitioning schemes). It is only when the final partition is reached that it becomes clear that the neighbourhoods differ substantially. All of the constraints can be changed by `moveTimeslotAndRoom` whereas `consistentSingleDaySwaps` only has interactions with the *eventClashes* and *threeInARow* constraints.

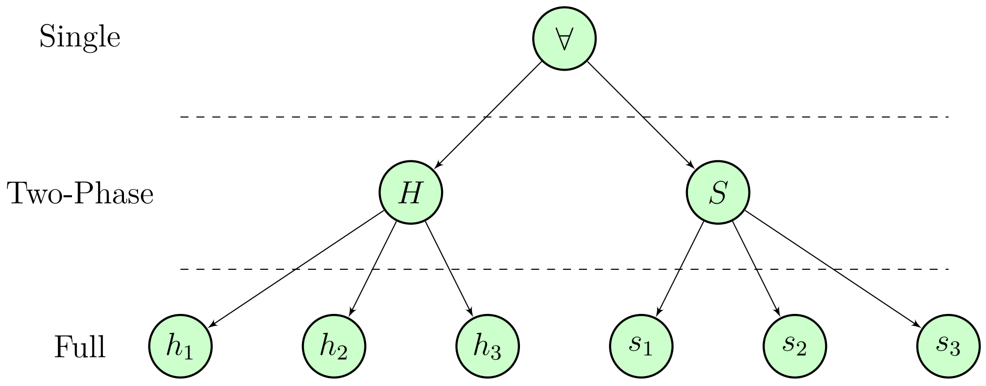


Figure 3.10: Tree of the detection results for `moveTimeslotAndRoom`.

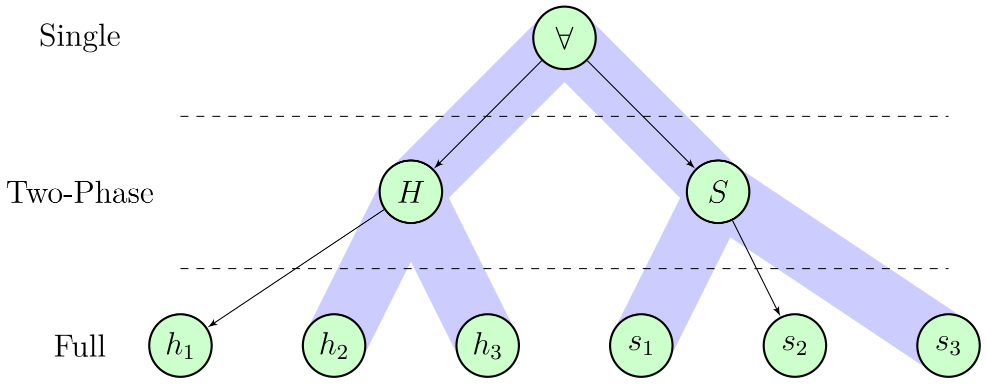


Figure 3.11: The detection results for `consistentSingleDaySwaps` displayed as a tree. The absent paths from `moveTimeslotAndRoom` are superimposed in blue.

Potential Efficiency Improvements

The results in Figs. 3.7, 3.8, and 3.9 show that automating the constraint interaction detection process is both achievable and accurate. Aside from the effectiveness it is worthwhile considering the efficiency of the detection process. Fig. 3.13 displays the time taken, in ms, to check each interaction. The results are split into two categories depending on whether or not an interaction was found. The time axis is displayed as a logarithmic scale due to the large range of values. At the lower times—those less that 10^2 ms—the results are grouped into clear bands. Timing in COMET, via the `System.getCPUTime()` method, is only accurate down to one millisecond. This exhibits itself as the distinctive banding when many runs map onto a relatively small range of potential times. Fig. 3.13 shows there is a clear difference between those runs where interactions were found versus those where none could be found. The *found* runs have a lower mean time of 2 s (to 169.6 s). There is tighter spread of times for the *found* runs as evidenced

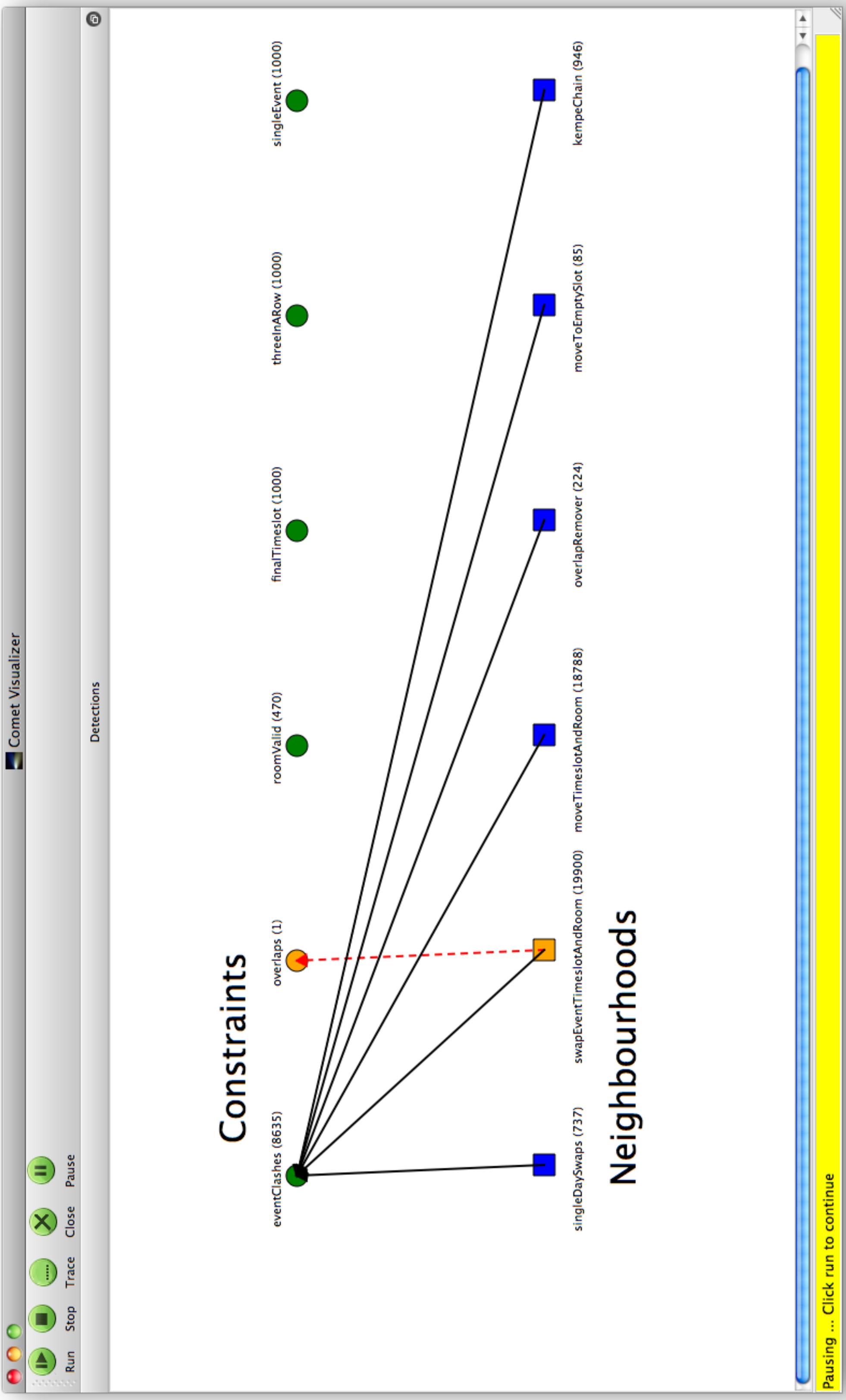


Figure 3.12: A screenshot of the Interaction Detector.

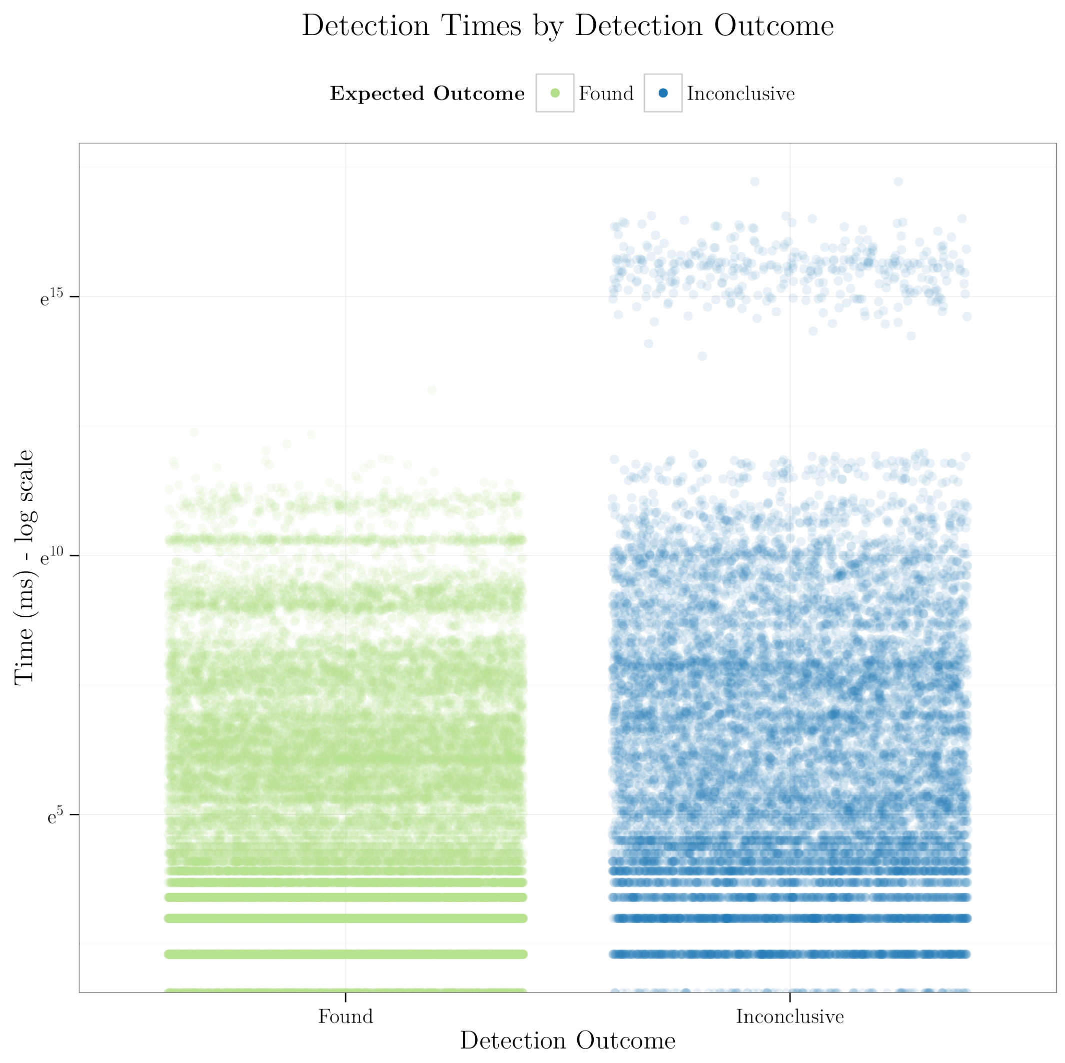


Figure 3.13: A box plot of the detection times of individual runs grouped and coloured by their detection outcome.

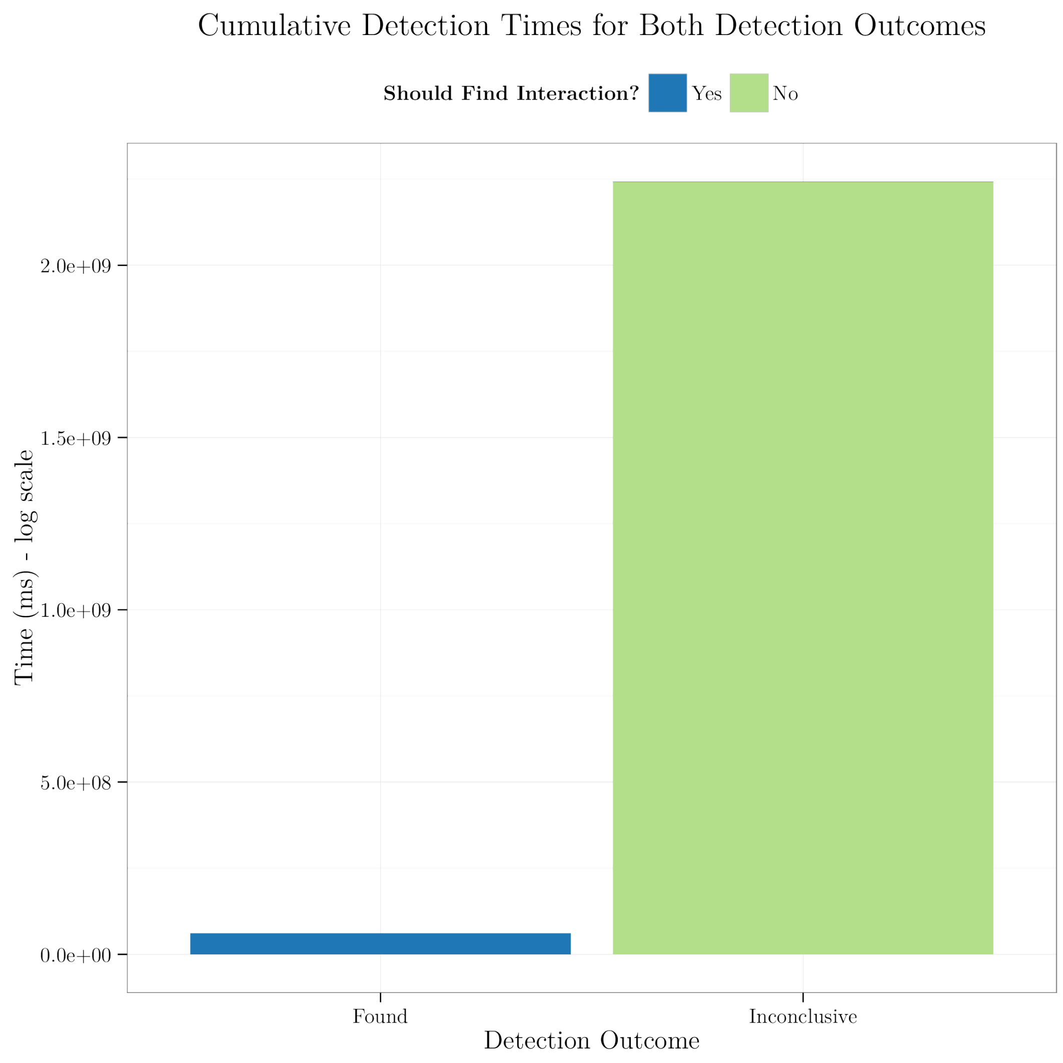


Figure 3.14: A comparison of the cumulative time spent searching for both expected and non-existent interactions.

by the standard deviation, σ, of 8.5 s versus 1120.4 s for the *inconclusive* runs.

As we stated in the Minor Caveats section, the Interaction Detector is incomplete; if it finds no evidence of an interaction it cannot conclusively state that none exists. This leads to situations where the detector is forced to exhaustively search a neighbourhood looking for non-existent interactions. Fig 3.14 shows that by far the largest part of the total detection time is spent searching for absent interactions. Out of all the runs 70.2 % are looking for existing interactions; however, these take only 2.64 % of the total detection time. To try to avoid, or at least reduce, this wasted effort we added an extra level of reasoning into the detector that allows it to recognise certain situations where interactions definitely *cannot* exist.

Detecting Absent Interactions

As covered earlier in the Neighbourhoods' Behaviours subsection of Section 3.1, we stated that there are three potential situations for the relationship between neighbourhoods and constraints: the projection of the closure of a neighbourhood's moves, $\pi_S(\mathcal{N}(A))$, is contained entirely within the constraint's relation, R; there is no intersection between R and $\pi_S(\mathcal{N}(A))$, or there is some overlap. To actually instantiate the relation of a constraint family would require creating the Cartesian product of each variable in the constraint's scope's domain. For a typical ITC instance with 200 events and 45 time-slots this would create 45^{200} states. Similarly, trying to create the closure of the neighbourhood would be impractical. The current detector operates by sampling both the closure and relation; this works well for identifying where relationships exist. However, the detector cannot recognise when no interaction exists because the closure is a subset of the constraint's relation. The same is true for when the closure and relation are disjoint except in one specific case: they are disjoint because the scopes are disjoint (i.e. the constraint is defined across different variables from those changed by the neighbourhood). This is the case that the detector extensions cover.

In COMET each incremental decision variable has a unique ID. Every `Constraint<LS>` (or `Function<LS>`) object allows the user to retrieve its scope (i.e. the variables it is defined across). Efficiently collecting the variables altered by a neighbourhood is less straightforward. We use the full exploration method but bypass the differentiation and **neighbor** sections, and instead set a boolean flag noting which variables had been selected. Since we are not exploring the neighbourhood with the intention of actually executing the move, it would be wasteful to invest time performing the delta calculations or creating and storing the move closures. Another minor optimisation is that the exploration prematurely stops when all the variables have been altered; after this point no additional information would actually be gained. When the full exploration is finished we can collect the IDs of those variables that were noted as having been changed.

If the set intersection of the constraint and neighbourhood variables is empty then it is a good indication that there could be no interaction between them. We cannot conclusively state that no interaction exists until we have checked for invariant dependencies. It may be that the constraint and neighbourhood variables are disjoint because the constraint has been defined in terms of an invariant. Figs. 3.15, 3.16, and 3.17 show the interactions uncovered by the enhanced detector; as with the original detector, there are no false positive detections and few situations where the detector fails to uncover interactions in all cases. In Figs. 3.16 and 3.17 only the hybrid Kempe Chain matching neighbourhoods have less than perfect detection rates.

The new spread of times—with additional possible outcome *No*—can be seen in Fig. 3.18. The new *No* runs have a mean time of 134 ms with a σ of 892.3 ms. The mean times for the found and inconclusive runs (1.9 s and 260.3 s respectively). The cumulative time investment for each classification is shown in Fig. 3.19; the No result is so small in comparison with the Found and Inconclusive runs that it appears as a thin line at the bottom. Out of the 17905 runs which did not result in an interaction being found, 35.4 % were proven not to have an interaction. This had the effect of reducing the total time taken to find inconclusive interactions by 0.66 % from 2 242 864.99 s to 2 228 065.62 s. The cost reduction is rather meagre, and it seems as if the additional time taken to collate the variable ids and check for invariant dependencies nullifies most of the efficiency gained by avoiding a full exploration.

Fig. 3.20 shows the distribution of the sum of the times for each constraint neighbourhood pair being investigated. The results were divided into two groups: those where an interaction exists, and those with no interaction. The *no interaction* results have a noticeably lower frequency than the *interaction* distribution. Also, there are two distinct small groups of outlying results at each end of the spectrum. Table 3.8 displays the top 20 constraint interaction pairs ordered by decreasing cumulative investigation time. Out of the 369 pairs the top 3 account for 94.9 % of the total experimental runtime. They are all two orders of magnitude greater

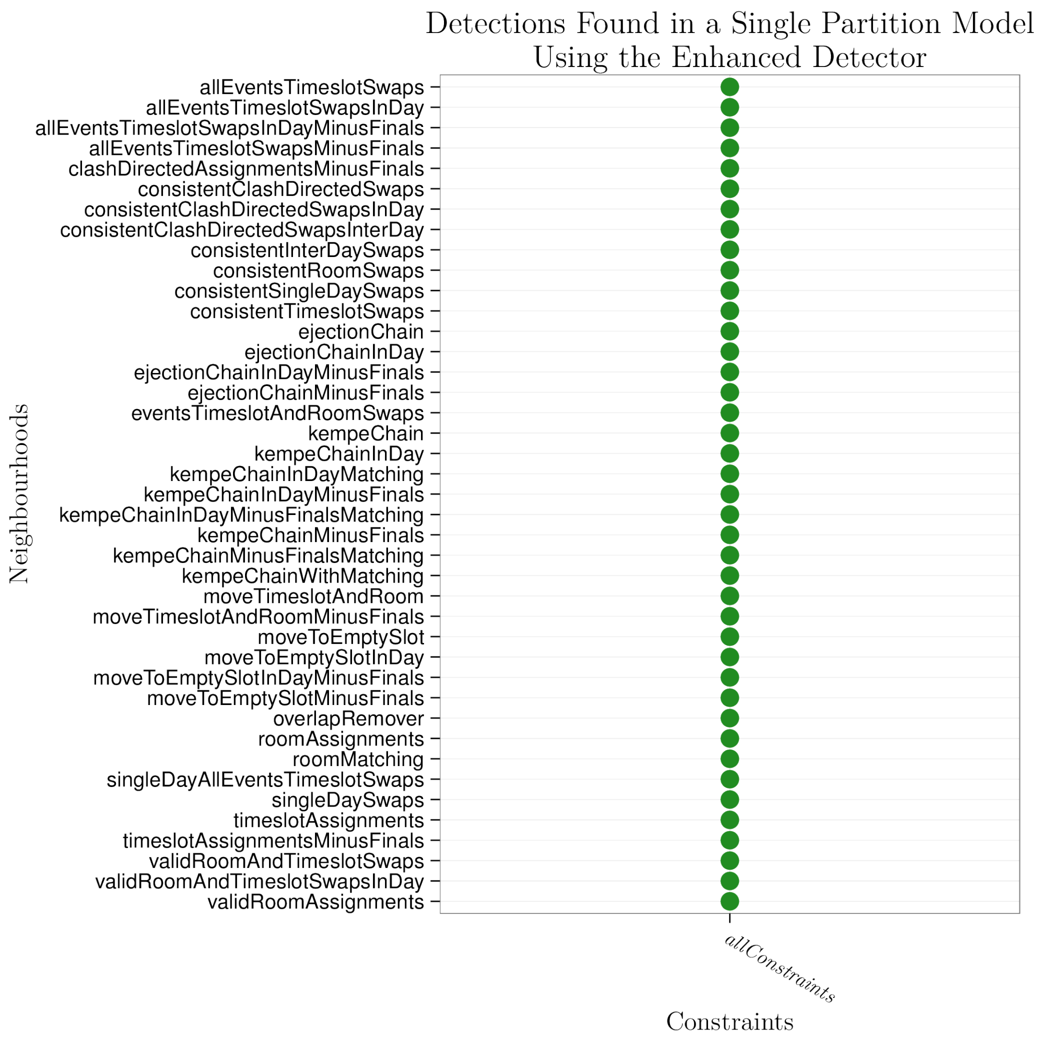


Figure 3.15: Enhanced interaction detections for a single partition model.

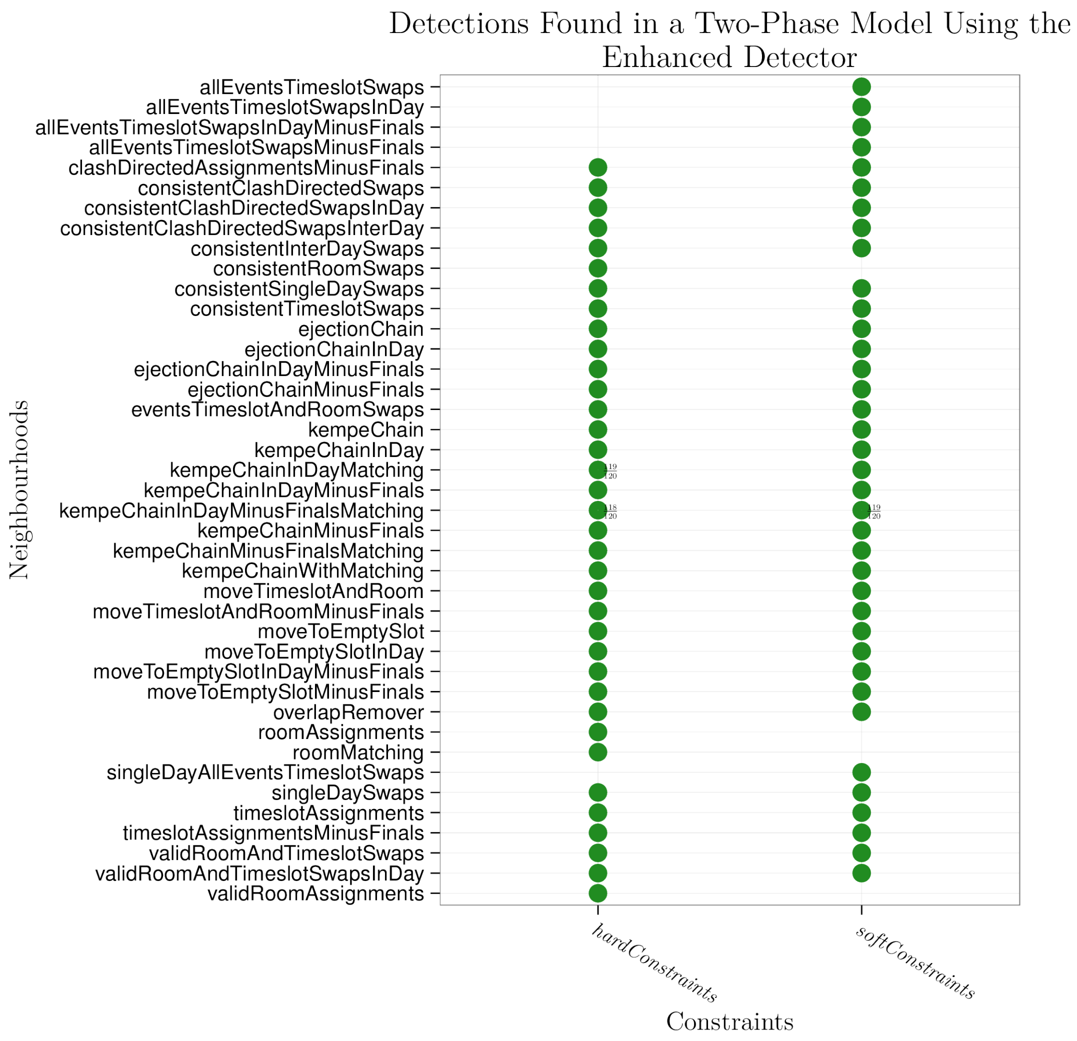


Figure 3.16: Interaction detections for a two-phase model.

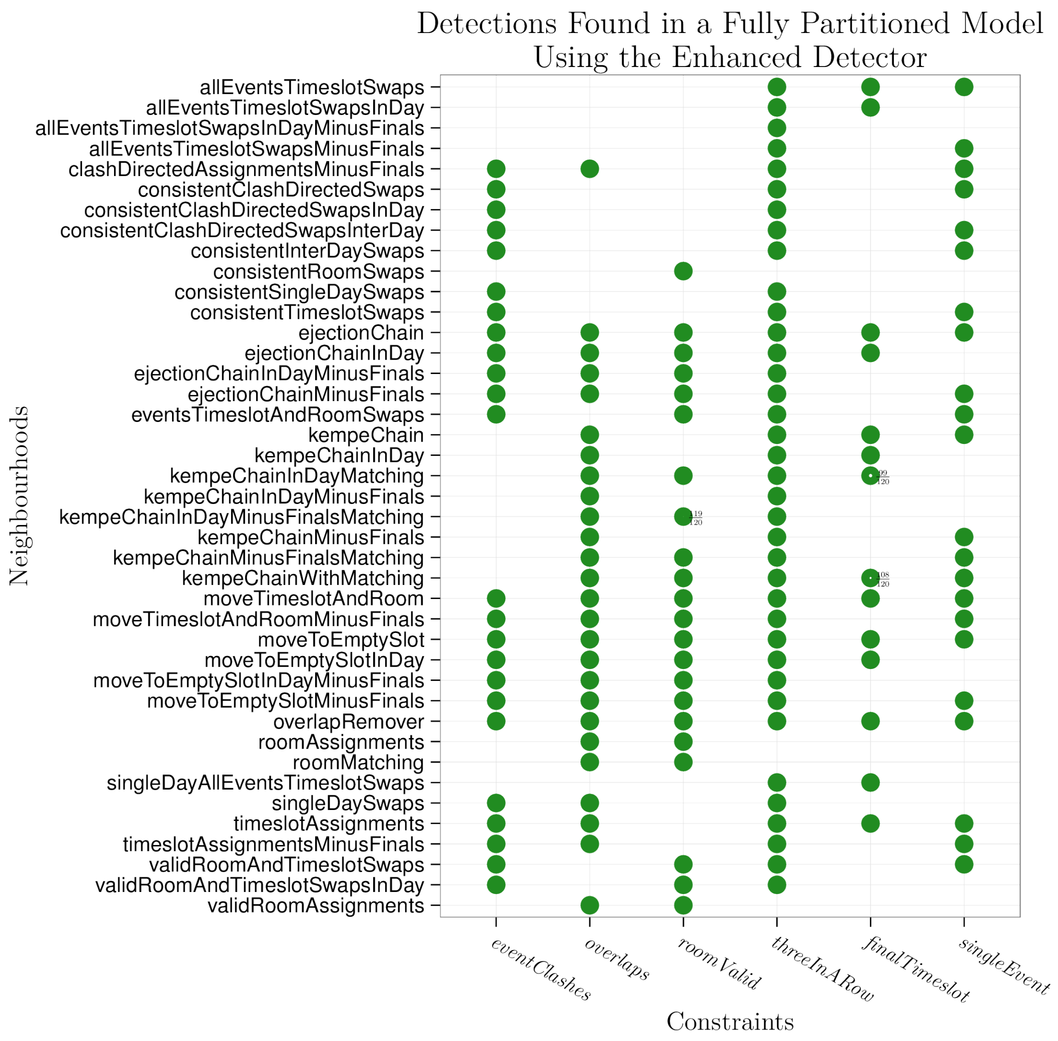


Figure 3.17: Interaction detections for a fully partitioned model.

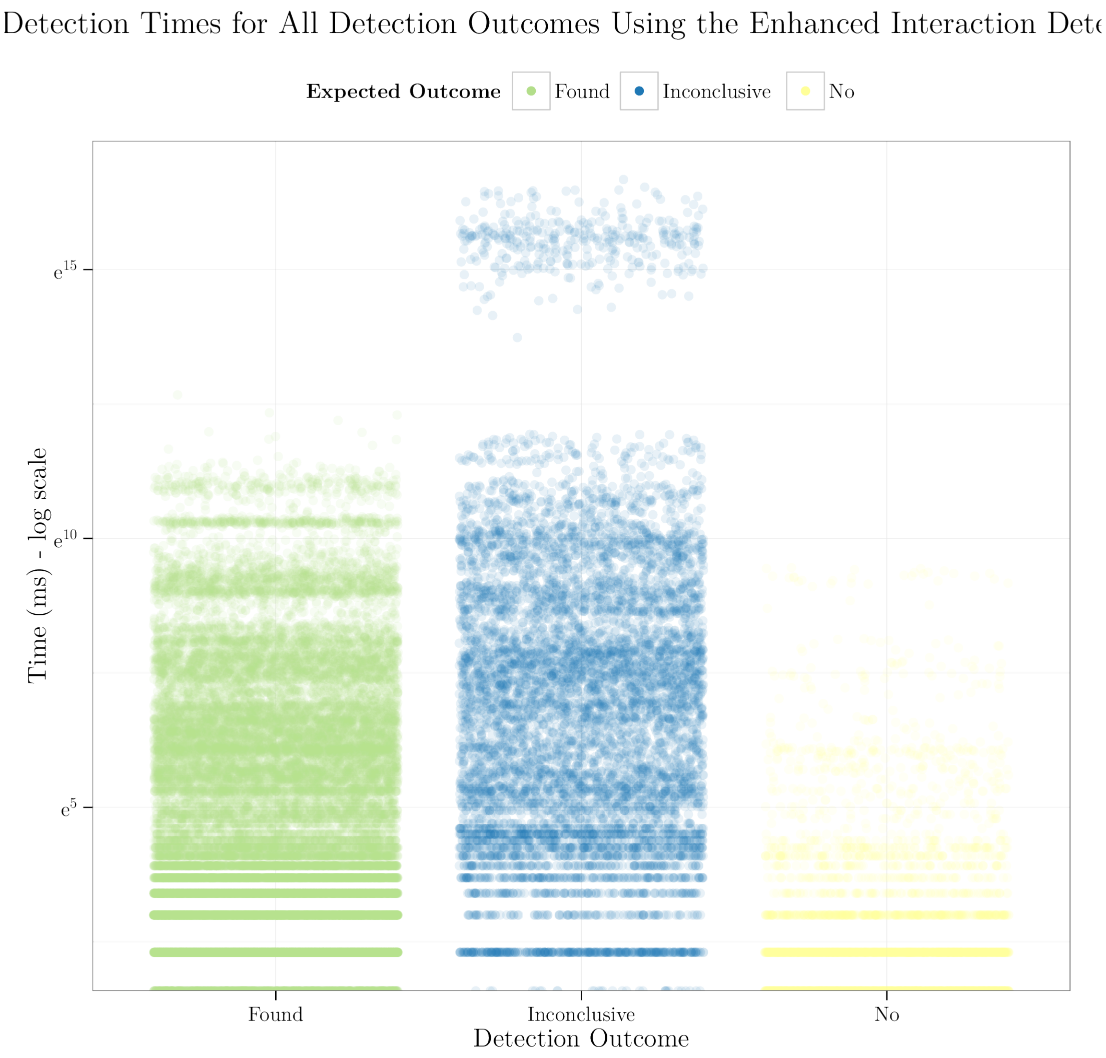


Figure 3.18: Detection times for the enhanced detector displayed by possible outcome.

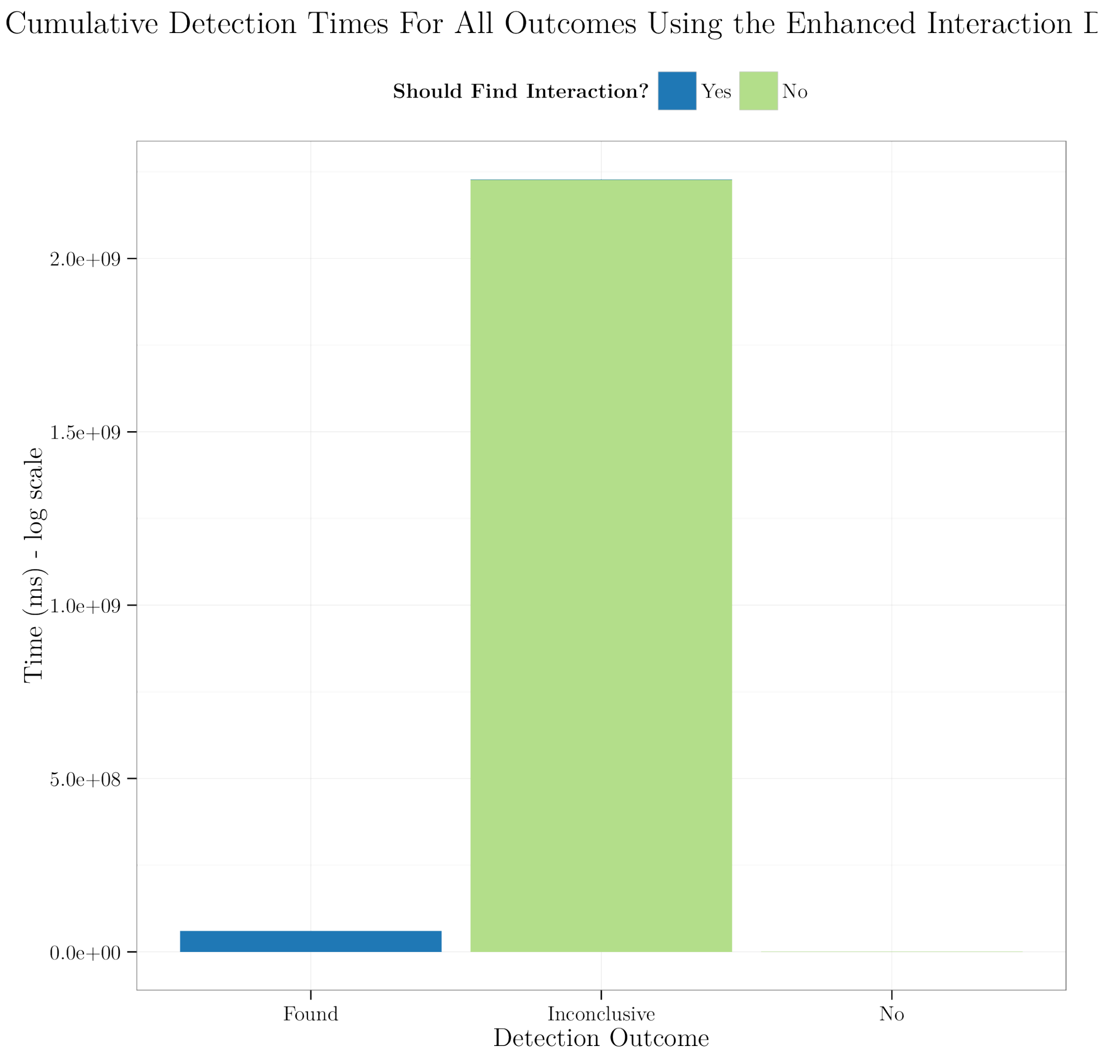


Figure 3.19: A comparison of the cumulative detection times.

Table 3.7: Comparing the Original and Enhanced Detector's performance over three model partitions with four metrics.

		Original	Enhanced
single	Accuracy	1.0000	1.0000
	Precision	1.0000	1.0000
	Recall	1.0000	1.0000
	Specificity		
hard-soft	Accuracy	1.0000	0.9996
	Precision	1.0000	1.0000
	Recall	1.0000	1.0000
	Specificity	1.0000	1.0000
full	Accuracy	0.9992	0.9988
	Precision	1.0000	1.0000
	Recall	1.0000	1.0000
	Specificity	1.0000	1.0000

than those pairs from fourth onwards. All three neighbourhoods are variants of the `EjectionChain<LS>`. In Table 3.6 the size of these neighbourhoods is shown as polynomial in the chain length, *cl*. For the experiments, all the chains were set to length three; hence, the neighbourhoods are cubic. If we omit the top three neighbourhoods from the calculations then the enhanced detector provides an improvement of 8.18 % reducing the total time to determine inconclusive results from 53 524.46 s to 49 146.43 s.

Assessing the Accuracy

The charts provide a clear visual way of displaying the detector's effectiveness at a constraint / neighbourhood level; however, they remain a somewhat subjective measure. For a more objective method of evaluating the detector, we turn to the tests used for *Information Retrieval* and *Categorisation systems*: precision, recall, specificity, and accuracy. Each constraint neighbourhood detection result is identified as one of four types: either *true positive*, *true negative*, *false positive* or *false negative*. True positive (*tp*), is where an interaction exists and was identified by the Detector. A true negative (*tn*) is where no interaction exists and none was detected. A false positive (*fp*) is where an interaction was identified where none exists. Finally, a false negative (*fn*) is where an interaction exists and was not found by the Detector. The formal equations for each of the assessment criteria

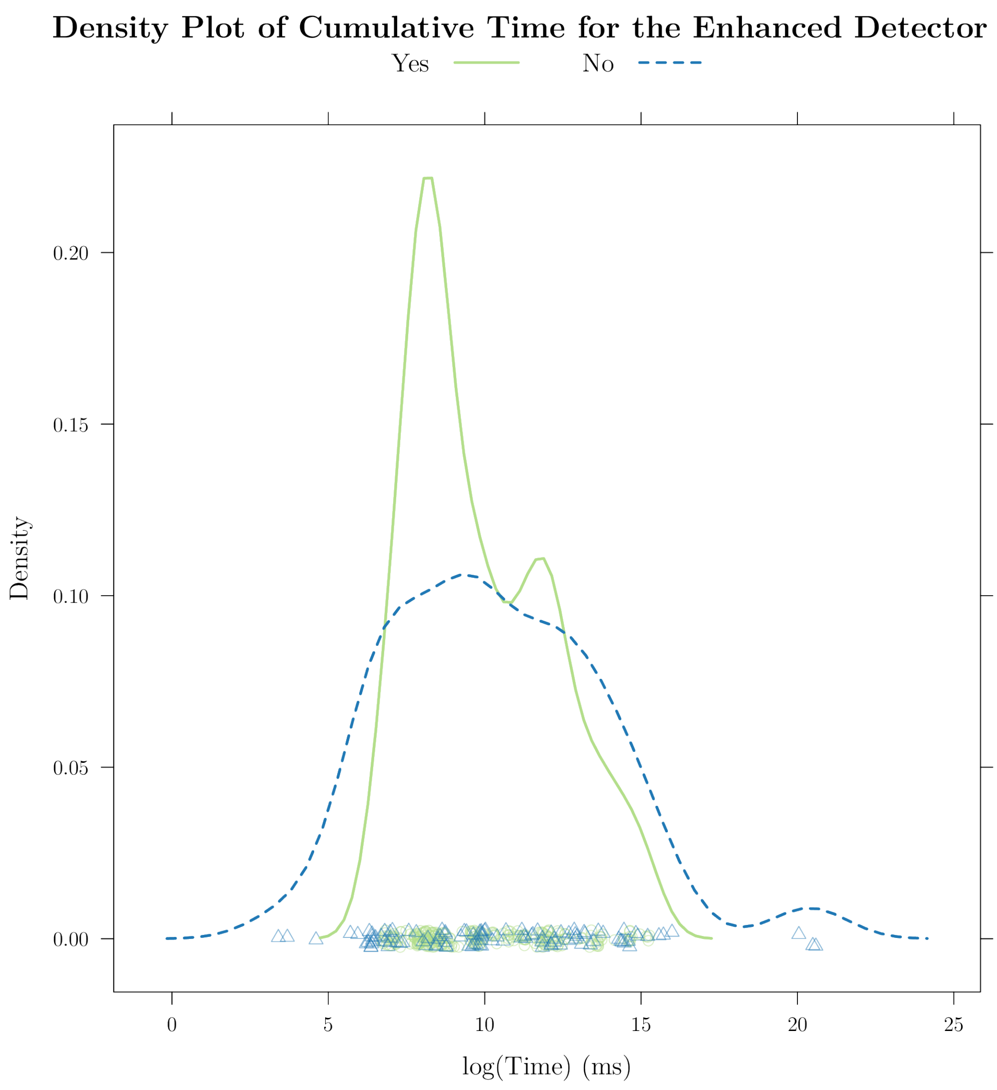


Figure 3.20: A density plot showing the log time distribution of all the constraint / neighbourhood pairs.

are:

$$\text{Precision} = \frac{tp}{tp + fp} \tag{3.9}$$

$$\text{Recall} = \frac{tp}{tp + fn} \tag{3.10}$$

$$\text{Specificity} = \frac{tn}{tn + fp} \tag{3.11}$$

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{3.12}$$

Table 3.8: The top 20 neighbourhood / constraint pairs (in terms of time used).

	Constraints	Neighbourhoods	Time (s)	Interaction	% of Total Time
1	*singleEvent*	`ejectionChainInDay`	873453.52	No	38.16
2	*finalTimeslot*	`ejectionChainMinusFinals`	792267.23	No	34.62
3	*singleEvent*	`ejectionChainInDayMinusFinals`	507413.95	No	22.17
4	*eventClashes*	`kempeChainMinusFinalsMatching`	8792.42	No	0.38
5	*eventClashes*	`kempeChainWithMatching`	6675.50	No	0.29
6	*finalTimeslot*	`ejectionChainInDayMinusFinals`	5784.49	No	0.25
7	*softConstraints*	`kempeChainMinusFinalsMatching`	4265.10	Yes	0.19
8	*allConstraints*	`kempeChainMinusFinalsMatching`	4124.10	Yes	0.18
9	*hardConstraints*	`kempeChainMinusFinalsMatching`	4061.50	Yes	0.18
10	*finalTimeslot*	`kempeChainMinusFinalsMatching`	4034.27	No	0.18
11	*overlaps*	`kempeChainMinusFinalsMatching`	3813.53	Yes	0.17
12	*overlaps*	`ejectionChain`	3146.27	Yes	0.14
13	*eventClashes*	`kempeChainMinusFinals`	2937.95	No	0.13
14	*eventClashes*	`kempeChain`	2765.19	No	0.12
15	*overlaps*	`ejectionChainMinusFinals`	2436.36	Yes	0.11
16	*eventClashes*	`moveTimeslotAndRoomMinusFinals`	2310.51	Yes	0.10
17	*hardConstraints*	`moveTimeslotAndRoomMinusFinals`	2289.08	Yes	0.10
18	*overlaps*	`moveTimeslotAndRoomMinusFinals`	2282.20	Yes	0.10
19	*allConstraints*	`moveTimeslotAndRoomMinusFinals`	2280.74	Yes	0.10
20	*softConstraints*	`moveTimeslotAndRoomMinusFinals`	2279.23	Yes	0.10

The *precision* of the detector (Equation 3.9) is a measure of how many of the detected interactions were valid and should have been uncovered. The *recall* or *true positive rate* (Equation 3.10) quantifies how likely the detector is to misclassify an interaction. The *specificity* or *true negative rate* (Equation 3.11) indicates how many of the situations where no interaction exists were correctly identified. The *accuracy* (Equation 3.12) is the proportion of results that were correct. Table 3.7 compares these four measures across all three model partitionings for both the original and extended detector. Because none of the detection runs generated any false positives, the precision and recall results remain fixed at 1.0. The false negatives introduced by the hybrid Kempe Chain matching neighbourhoods can be seen affecting the accuracy slightly in the *hard-soft* and *full* partition results. However, even with these false negatives the accuracy remains above 99 %.

3.7 Reusing Detections

The interaction detection process can be automated but may require a significant investment of time. The relationships that are uncovered from analysing one instance are transferable to any other instance of the same problem. We are assuming that for most problems there is not a significant change in their properties. For example, a timetabling instance could have events with no registered students, thus would not cause clashes when moved between time-slots; however, this seems unrepresentative of real-world problems. If detections are transferable then the detection process can be performed off-line and the results cached for future reference. The detection results can be written to simple flat text file format (shown in Fig. 3.21). The benefit of a plain-text format is that it still allows the user to provide detection information if they wish.

```
<# of Constraints>
<# of Functions>
// One line per Constraint
<constraint id> <constraint name> <list of the interacting neighbourhood ids>
// One line per Function
<function id> <function name> <list of the interacting neighbourhood ids>
// One line per neighbourhood
<neighbourhood id> <neighbourhood name> <size>
```

Figure 3.21: The syntax used to cache the detection results.

3.8 Conclusions

This chapter set out to describe how information about which constraints a neighbourhood could affect could be detected automatically. We defined this concept as *constraint-neighbourhood interactions*. Previous approaches required either human intervention or used a specialist problem representation. Our system is built on top of the CBLS language, COMET, and uses a model representation that should be familiar to anyone who has used other common CP systems (such as OPL or ILOG Solver). We have described the creation of a framework that allows the interaction detection process to be performed in an entirely non-problem-specific manner. We assessed the technical feasibility of this and found that the runtime and memory costs were negligible. We have also introduced several combinatorial problems: the PPP, Graph K-Colourability, and Post Enrolment-based Timetabling. The constraint-neighbourhood interaction detector was evaluated on the timetabling problem. The results were accurate and uncovered all the relationships. We also investigated whether some basic reasoning could improve the system's efficiency and how we can cache the detection results for future runs.

We have reached the stage where we can automatically uncover information about how search neighbourhoods will behave against a particular constraint model / partitioning. This is an important step towards bringing CBLS closer to other

AI disciplines such as Automated Planning. Currently, Local Searches apply a function to an existing configuration, assess the quality of the resulting neighbours and then select from amongst them to progress the search. The neighbourhood appears as a *black-box* to the algorithm; the designer may understand its behaviour but the algorithm does not. Central to AI planning is the idea that goals can be satisfied by altering the world through the application of predictable actions. Each action has a clearly defined set of preconditions and effects. If the current state of the world satisfies an action's preconditions then the application of that action will cause the effects. Our constraint-neighbourhood interaction analysis moves CBLS towards being able to view neighbourhoods as actions (in the planning sense) that have definite behaviours. The partitioning of the constraint model into families gives a set of facts about the state of the world against which we can classify the behaviour of a neighbourhood.

The next chapter investigates some of the ways that we could potentially use the partitioned models and interaction information to create more dynamic searches.

Chapter 4

Exploiting Interaction Information

Every man is wise when attacked by a mad dog; fewer when pursued by a mad woman; only the wisest survive when attacked by a mad notion.

Samuel Marchbanks' Almanack
W. Robertson Davies

Chapter 3 introduced the concept of *constraint-neighbourhood interactions* and presented a framework written in Comet for detecting these interactions in an automated, problem-independent fashion. This chapter explores how the increased information provided by constraint partitionings and constraint-neighbourhood interactions can be used within search procedures.

Decision making could be described as a spectrum. At one extreme is Local Search which makes short-term decisions based upon the current search position and the surrounding neighbours; from the incumbent solution it generates some neighbours, evaluates them, and then selects the *best* according to some criteria—which may be manipulated (as in SA, TS, etc.). CP represents the other end of the spectrum; the repercussions of each search decision are explored. The level of consistency enforced on the constraints determines how much information is propagated. However, CP remains a methodical approach that tries to reason about the effects of each selection, "*CP = Search + Inference*". CBLS augmented with constraint-neighbourhood interaction information has the potential to sit somewhere between these two schemes. At its core, CBLS still retains Local Search's short-term selection policy; it could, however, take a more informed view and include some reasoning over higher-level search direction decisions such as which neighbourhood to select at a given point. It is this variety of inference that we hope to give CBLS with the extraction of constraint-neighbourhood interaction information.

Section 4.1 investigates whether interaction information can be used within a VNS to reduce unnecessary exploration. To achieve this we represent the internal dynamics of a VND algorithm as a Generalised Local Search Machine (GLSM). Using a GLSM allows for a more dynamic and reactive control flow; one where the configuration of a VNS responds to the current search state. In VNS the neighbourhood ordering is static and increasing in size; Section 4.2 explores whether there is any benefit in retaining a linear increasing ordering that is updated at each iteration to reflect changes in neighbourhood sizes. Section 4.3 expands further upon the idea that the neighbourhood ordering can change at runtime by introducing an algorithm, CDVNS, where the neighbourhood ordering is derived from a constraint ordering. Lastly, Section 4.4 looks at how interaction information can provide modelling and search configuration feedback. The aim of this chapter is to cover a broad selection of the ways in which constraint-neighbourhood information can be applied in a search context.

4.1 Enhancing VNS with Optional Transitions

In Section 2.10 we covered Mladenović and Hansen's VNS framework where a search has multiple neighbourhoods at its disposal. The neighbourhoods are explored in a linear order (in the case of VND, the most common form of VNS). This order is defined before the search is initiated and typically arranges the neighbourhoods by increasing size. As a VND search progresses, if it fails to find an improving solution the next neighbourhood in the sequence is explored. When an improving solution is discovered the search continues from the first neighbourhood in the ordering.

GLSMs are a formalism developed by Hoos and Stützle [2005, Chp. 3, p. 113] to concisely describe the interactions between the components of hybrid search algorithms. Each component becomes a node within a Finite State Machine (FSM);

the transitions between the states can be either: unconditionally deterministic (DET); unconditionally probabilistic (PROB(p)), where p is a given probability; conditionally deterministic (CDET(C)), where C is a logical condition; or, lastly, conditionally probabilistic (CPROB(C, p)). The logical conditions used for CDET and CPROB take the form of simple predicates. Table 3.1 in Hoos and Stützle [2005, Chp. 3, p. 126] lists some recurrent examples. The GLSM notation was intended for describing high-level couplings between abstract search components (e.g. constructive phase, disruption phase, random walk, etc.); however, it is also well suited for describing the low-level configuration of neighbourhoods within VND algorithms. In Fig. 4.1 there is an example VND configuration for an algorithm with five neighbourhoods, $N_1, \ldots, N_5$. The transition between one neighbourhood and its successor is conditionally deterministic and occurs when no improving solution is found. Using Hoos and Stützle's conditions, this would be equivalent to their predicate CDET(noimpr(k)) with $k = 1$, henceforth denoted CDET(I) for brevity. If at any stage an improving solution is found the search returns to the first neighbourhood, i.e. the conditionally deterministic transition CDET($\neg I$). The GLSM succinctly describes the behaviour of the VND algorithm; it bears a strong resemblance to a *transitive closure* graph.

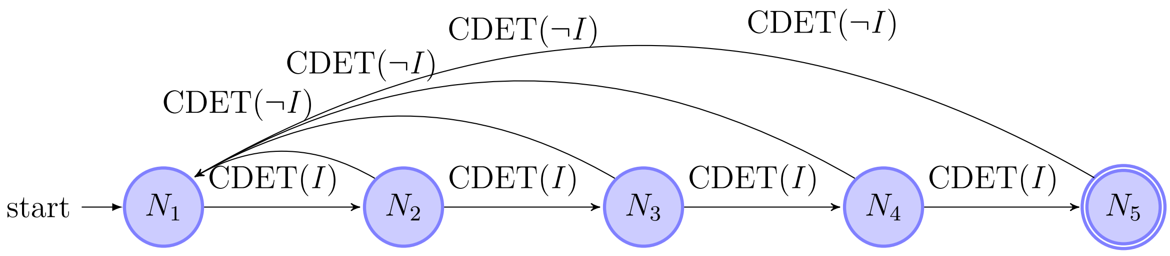


Figure 4.1: A basic VND represented as a GLSM.

The disadvantage of a linear search progression is that each neighbourhood has to be explored—and shown to be non-improving—before moving to the next in the sequence. What this assumes is that all the search neighbourhoods have the same effects and that there is no way to determine the current state of search. However, as we saw in the previous chapter this is not necessarily the case; it *is* possible to efficiently maintain information about the violations of multiple constraint families and accurately predict the effects of neighbourhoods on those families. Constraint-neighbourhood interaction information can be used to identify when a neighbourhood will not interact with unsatisfied constraints. Rather than exploring the initial neighbourhoods repeatedly, the interaction information could be used to bypass provably useless neighbourhoods. Fig. 4.2 shows the new configuration of the algorithm with additional transitions (drawn with dashes) that can be used when the current neighbourhood is not *applicable*. The transition condition is denoted as CDET($\neg A$).

Curtois et al. [2006] make similar alterations to a VNS where neighbourhoods can be omitted at run-time. Their work attempts to balance the intensification and diversification of a VNS by creating a mathematical model of a neighbourhood's behaviour and using this model to identify what they describe as *bad* neighbourhoods. The model is updated as the search progresses and attempts to predict whether a neighbourhood will lead to an overall improvement before searching it. Their notion of the behaviour of a neighbourhood makes no reference to the actual properties of its permutations or the structure of the problem.

4.1.1 VNS as a GLSM

Creating a VNS from a GLSM is relatively straightforward given that our framework treats neighbourhoods as first-class objects (e.g. they can be function arguments or return values). As outlined in Section 3.3 neighbourhoods are objects, sharing a common interface, that are collected together into a contiguous array. A search does not require neighbourhoods to be hard-coded into it; they can be passed as an argument at run-time. The behaviour of a VNS algorithm can be described as the manipulation of an index variable. If exploring the neighbourhood at a given index does not yield an improvement then the index is incremented for the next iteration. When an improving solution is found the index is reset to its starting value. Listing 4.1 shows the COMET code used to express the VNS in our experiments. The code only expresses the transition rules for a VND; there are no explicit references to any particular neighbourhood, nor is there a fixed ordering. The ordering is stored as a graph where nodes correspond to the indices of neighbourhoods within the framework's array. This means that the VNS in Listing 4.1 can be used for a collection of neighbourhoods; indeed, during the evaluation five different sets of neighbourhoods are used.

4.1.2 Experimental Evaluation

To ascertain whether these additional transitions can aid search performance (by reducing wasted exploration), we conducted an experiment using the timetabling

```
void searchVNS(bool useRestarts, bool skipInvalid) {
  cout << "searchVNS: " << _iterations << endl;
  _useRestarts = useRestarts;
  _skipInvalid = skipInvalid;
  assert(hasNeighbourhoodOrdering());
  updateNeighbourhood(getFirstNeighbourhood());
  position = new Integer(1);
  if(useRestarts) {
    setupRestartThresholds();
  }
  setupVNSSearchFails();
  if(_searchStart == 0) {
    // If VNS is a component the outer search sets this.
    _searchStart = System.getCPUTime();
  }
  while(!_shouldStop && hasTimeLeft()) {
    if(_refreshSizes) {
      updateNeighbourhoodSizes();
      _scheduler.setNeighbourhoodSizes(_neighbourhoodSizes);
      setNeighbourhoodOrdering(_scheduler.
   getLinearNeighbourhoodOrdering(true));
    }
    Neighbourhood<LS> n = getNeighbourhood();
    if(shouldSearchNeighbourhood()) {
      cout << "Exploring " << getNeighbourhoodName() << " (" <<
   getNeighbourhoodSize() << ") via " << _strategy << endl;
      int pre = System.getCPUTime();
      search(n, N);
      moveTime := System.getCPUTime() - pre;
      if(N.hasMove() && shouldAccept(N.getIntMin())) {
        cout << "Executing: " << N.getIntMin() << endl;
        call(N.getMove());
        displayProgress();
        // If found move, snap back to the first Nhood
        if(!isFirstNeighbourhood()) {
          resetVNS(position);
        }
      } else {
        updateStagnantIterations();
        if(hasExhaustedVNS()) {
          cout << "Exhausted neighbourhoods and restarts" << endl
   ;
          _shouldStop := true;
        } else {
          vnsFailedSearch++;
          advanceToNextNeighbourhood();
        }
      }
    } else {
      vnsOmits++;
      cout << "Omitting fruitless neighbourhood" << endl;
      advanceToNextNeighbourhood();
    }
  }
}
```

Listing 4.1: The VNS code from `OriginalSearch.co`.

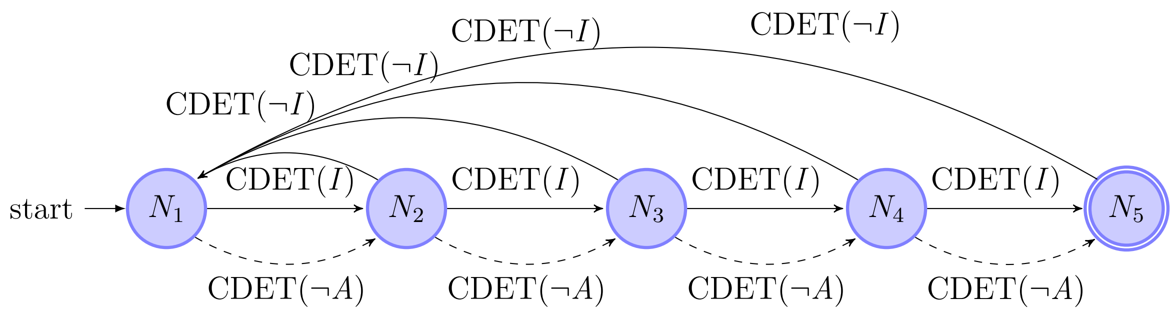


Figure 4.2: The GLSM for a VND where irrelevant neighbourhoods can be bypassed.

problem from the previous chapter. The performance of a VNS for a given problem will be dependent upon the neighbourhoods available to it. For the detection experiments we created 41 neighbourhoods; far beyond the typical set of between four and six that other timetabling solvers use. The largest collection of neighbourhoods that we are aware of appears in the Nurse Rostering Problem work of Burke et al. [2010]. They have a collection of 23 neighbourhoods from which they select a subset using a GA.

Selecting the "*best*" set of neighbourhoods for the Timetabling Problem is not the focus of this work; however, any experimental results will be biased by that selection decision. To counter this we created five distinct sets of neighbourhoods (`nhoods1.txt`,...,`nhoods5.txt`) that are shown in Table 4.1 (page 143). We do not make any claims about the particular effectiveness of any of these sets. The union of the interactions for each set comprises the full set of constraints (i.e. each configuration can affect all the constraints). The neighbourhoods were ordered linearly by increasing size (based upon their size at an instance's initial solution). These linear orderings were given to the VND algorithm in Listing 4.1 that searched each neighbourhood with a *first improvement* acceptance strategy. For each configuration (i.e. omitting neighbourhoods or not), we ran the search 10 times on each of the 20 ITC problem instances. To ensure a fair comparison, every run for a given instance started from the same initial configuration. Table B.2 (page 232) shows the quality of each starting solution. The interaction information was loaded from a cached file rather than being detected before each run. The experiment was distributed using GNU Parallel [Tange, 2011] across 55 PCs. All the machines were running COMET 2.1.1 on Ubuntu 12.04. Thirty machines had 2-core 3.2 GHz CPUs and 8.0 GB of RAM; the remaining 25 machines had 4-core

3.2 GHz CPUs and only 4.0 GB of RAM. Whilst using GNU Parallel we shared the configurations such that the 2-core machines would only have a single configuration running at any given time; we allowed two concurrent experiments to run on the 4-core machines. We only investigated two different model partitionings: the hard and soft constraints, and the full partition. Having only a single constraint family provided no information that would allow the search to omit neighbourhoods; every neighbourhood displays the same interactions (as evidenced in Figs. 3.7 & 3.15).

Each run was subject to two restrictions: the total run-time should not exceed seven minutes, and when the search reaches a local optimum it terminates (i.e. there is no disruption phase or SA-style acceptance of worsening moves). The best solution reached when a run completed was passed to the ITC solution validator. The violation scores plotted in the results are those returned by the official validator. Using the validator's result creates a static benchmark against which our solutions would be consistent with those generated by any other competitor; it also means that the CSP model does not need to exactly match the problem definition. The seven minute time cut-off was determined by running the ITC machine benchmarking application on one of the experimental machines. At the time of the competition the limit was intended to be around fifteen minutes; however, the increase in power of modern machines has brought this down. A minor caveat is that the neighbourhood exploration phase of the search is not interruptible. If the time limit expires whilst the search is in the midst of an exploration, it will continue until either it finds a solution, or has to switch neighbourhoods. At this point it will realise the time has expired and stop any further searching.

Presentation of the Results

The results of the experiments in this chapter are all displayed in the same format that warrants some discussion. After reaching the termination criteria—and being processed by the solution validator—a run has a violation value associated with each of the six constraint families. For the initial experiments only the total violations (e.g. the summation of the families' violations are shown). In later plots which display the result on a constraint family basis then the six families appear as separate facets (e.g. sub-plots) that are arranged vertically. There are typically two (though occasionally three) columns of constraint facets.

Primarily the columns show the partitioning of the model (e.g. Hard-Soft, or Full). Some later experiments use only a single partitioning, in which case the columns represent the search type or some other configuration choice. These are clearly labelled on the figures and also in the text discussing the results in the relevant sections. At the bottom of each column is a plot showing the total violations (e.g. the sum of all the violations from the families in that particular partition).

The x-axis of each plot shows the remaining violations at the point at which the run terminated. The y-axes display the individual problem instances. The leftmost column shows the instance name; to allow for ease of scanning (and to save space) the subsequent columns omit this. Each row has a prominent red 'x' marking the starting violations for that instance.

The main visual cues used in the plots are *colour* and *symbol shape*. The colour designates the variable under investigation; for example, in the experiments for Section 4.1.2 the colour indicates whether or not bypassing non-applicable neighbourhoods is allowed. The symbol shape displays which of the neighbourhood configurations (shown in Table 4.1) the run used. Each plot has a legend at the top specifying what the colours and shapes represent (though the shape's meaning is effectively constant). The colours are opaque to make dense areas of values clearer and mitigate against over-plotting.

One aspect of the column structure that should be kept in mind is that whilst the constraint families are displayed individually, the search may not be able to "*see*" such a fine-grained distinction. In the hard-soft partitions the top three constraints (*eventClashes*, *overlaps*, and *roomValid*) will only be visible to the search as a single *hardConstraints* family (similarly the bottom three facets are the *softConstraints*). This can produce some potentially unexpected results. Any reasonable search should be able to find solutions that distribute points closer to the x-axis origin than the initial starting solution (e.g. to the left of the red 'x'). However, points falling to the right of the initial value do not mean the search has actively degraded that constraint; the search is not aware of the changes it causes to the constraint violations at a separate family level. We already touched on the disparity between the information gained from different granularities of partitionings in Section 3.2 of Chapter 3 (and represented it visually in Fig. 3.10).

Statistical Tables

The plots aim to provide a clear visual representation of the results for the various searches. Whilst it is possible to see trends and clusters emerging, only a statistical test can determine whether these are *significant*. There are numerous statistical tests available, but for assessing our results we use the Wilcoxon Sign Rank Test (also known as the Mann-Whitney test). The Wilcoxon Sign Rank Test allows us to compare two different sets of results and decide whether the differences are statistically *significant*. The Sign Rank Test is non-parametric which means that (unlike other tests such as the paired t-test) it does not make any assumption about the distribution of the results. The paired t-test requires the data to be *normally* distributed (i.e. in a Gaussian distribution). The Wilcoxon Sign Rank test is *paired*; this makes it appropriate for the situations where multiple samples (under the same conditions) are made for both techniques under comparison. We show the results of the Wilcoxon Sign Rank Test for each experiment in their own table. Each table shows the p value, the Wilcoxon Sign Rank test statistic value, W, and the confidence interval. We used the standard 95 % level of confidence; any p values of less than 0.05 indicate a significant result. Values that fall below the 99 % and 99.9 % levels are also highlighted. If the p value is beneath the significance threshold then we can reject the Null Hypothesis, H_0. This allows us to say that (with a 95 % confidence) that the results do not come from the same distribution. However, for our tests we use the slightly stronger *one-tailed* assumption, H_1. This expands upon H_0 by adding a direction; for example, not only are the results A and B not from the same distribution, but the mean value of A is greater than B. This allows us to test whether one algorithm (or configuration) offers a statistically significant *improvement* over another. We only display the lower bound of the confidence interval $\bar{x} - E$; the upper bound is not shown because (for the one-tailed test) it does not have any meaning (and is assumed to be positive infinity). The test statistic W value is what you would use to look up the significance value in a statistical test table (though the p value makes this unnecessary). Another verifier of statistical significance is if 0 does *not* appear between the lower confidence interval and infinity; for example, $\bar{x} - E = -25$ would not be significant (because $-25 < 0 < \infty$) whereas $\bar{x} - E = 85$ would be.

Using a linear neighbourhood ordering has implications for the number (and distribution) of *failures* (i.e. explorations where the neighbourhood did not find an improving solution). To reach a later neighbourhood an earlier one must have failed. Consequently, the initial neighbourhood has more failures than the second and so on. There can only be a single failure in the final neighbourhood; if the last neighbourhood fails to find an improving solution then the search concludes that it has reached an optima and terminates. The earlier neighbourhoods will experience more failures, but they will also be smaller. Whether multiple failures of small neighbourhoods are more costly than fewer failures of larger neighbourhoods is not clear. Our expectation is that the early failures will cumulatively represent a significant time cost. However, the relative sizes of the neighbourhoods in a given configuration are bound to influence this.

4.1.3 Results

The results of the experiment are shown in Fig. 4.3. The intention was to evaluate whether allowing the omission of neighbourhoods improved performance. We quantify performance by using the total number of violations in the best solution the search returns. We hypothesise that bypassing non-applicable neighbourhoods will reduce redundant exploration and allow further exploration within a given amount of time. However, simply allowing the search more iterations is not in itself interesting unless that also translates into finding better solutions. Therefore, we chose to represent the performance using the number of violations in the best solution found.

Figure 4.3 displays several interesting properties of the experiment's results. Firstly, all the runs appear to the left of the initial starting solution. This means that the search was always able to move towards an improving solution. However, none of the results manage to reach the optimal solution (of zero violations). Runs do reach around 300 violations in many instances. It is also apparent that the neighbourhood set used in a run strongly influences the quality of the solution where search terminates; the plot clearly shows clusters of the same symbol shape (i.e. the same neighbourhoods). What fails to emerge is any real distinction between the solutions reached when bypassing neighbourhoods or not. The clusters all contain a mixture of colours. This holds between the two partitioning schemes. Results in the fully partitioned model do appear to be more separated between those which find large improvements and those that only improve slightly.

We applied the one-tailed Wilcoxon Sign Rank test to results provide a more empirical analysis of the data in the plot. These results are displayed in Table 4.3.

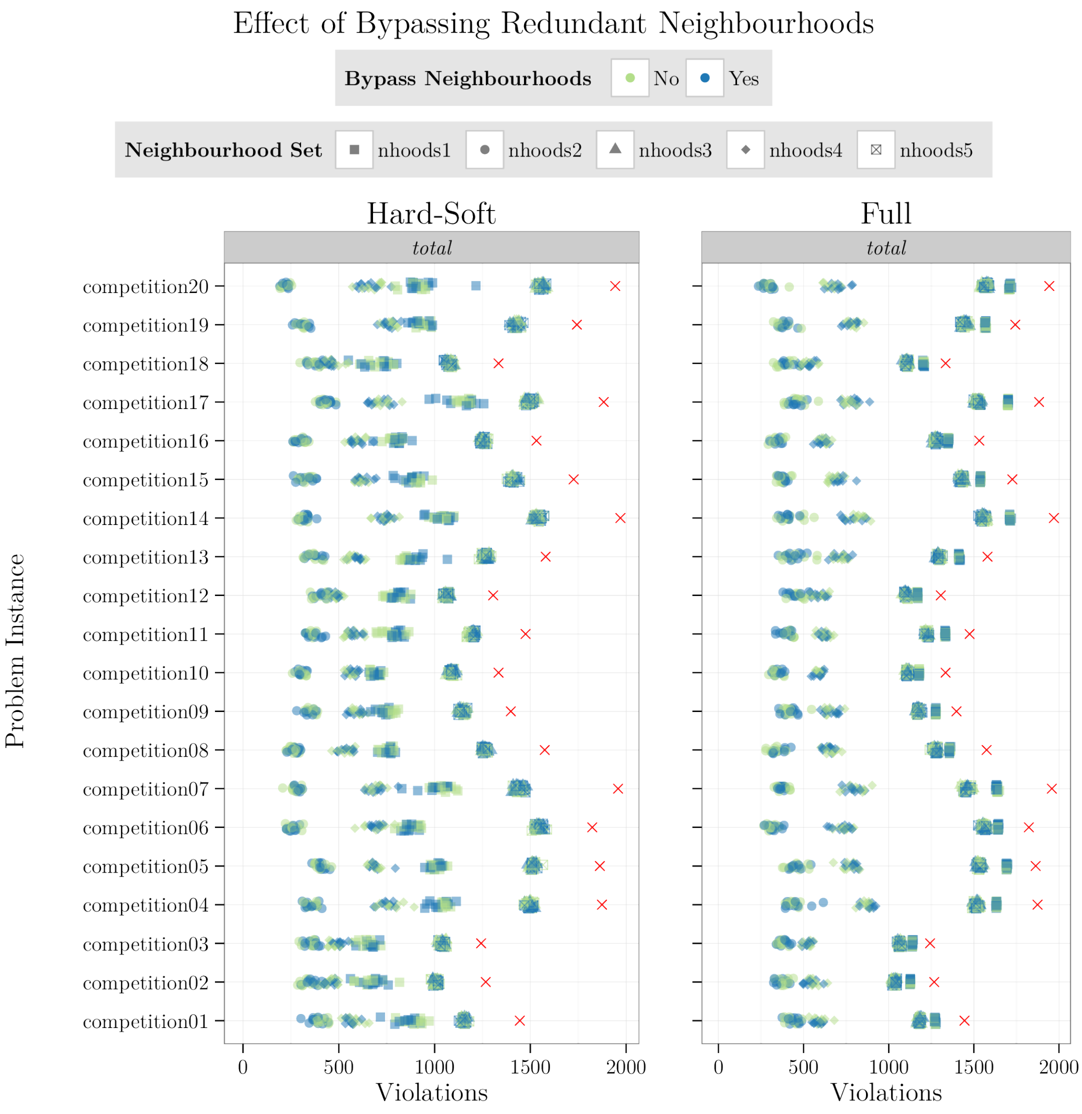


Figure 4.3: Comparing the effect of allowing additional transitions in the VNS on the competition instances.

Table 4.1: Five neighbourhood configurations used to compare the searches.

Neighbourhood	`nhoods1`	`nhoods2`	`nhoods3`	`nhoods4`	`nhoods5`
`allEventsTimeslotSwaps`			✓		✓
`allEventsTimeslotSwapsInDay`	✓				
`clashDirectedAssignmentsMinusFinals`		✓			
`consistentClashDirectedSwapsInDay`	✓				
`consistentClashDirectedSwapsInterDay`	✓				
`consistentInterDaySwaps`				✓	✓
`consistentRoomSwaps`			✓		✓
`consistentSingleDaySwaps`				✓	
`consistentTimeslotSwaps`		✓			
`ejectionChainInDayMinusFinals`					✓
`eventsTimeslotAndRoomSwaps`					✓
`kempeChainInDayMinusFinalsMatching`					✓
`kempeChainWithMatching`		✓			
`moveTimeslotAndRoom`				✓	
`moveToEmptySlotInDay`			✓		
`overlapRemover`	✓			✓	
`roomAssignments`				✓	
`roomMatching`		✓			
`singleDayAllEventsTimeslotSwaps`			✓		✓
`singleDaySwaps`			✓		
`timeslotAssignments`				✓	
`validRoomAssignments`	✓				

They reiterate what emerges from Fig. 4.3: no statistically significant improvement is gained from allowing the bypassing of non-applicable neighbourhoods.

Further Investigation

Allowing the search to avoid exploring provably redundant neighbours would seem like it should be beneficial. However, the results do not support this. The solutions reached do not differ significantly in terms of quality. To try to address why this may be the case we performed some additional analysis of the runs. Figure 4.3 shows that all the runs find improving solutions, so the search is definitely progressing through the search space. The search always starts from the same initial solution for a given instance, and that starting solution will have an associated level of constraint violations. The pattern of which constraint families are violated (and which are satisfied) is the *signature* which determines whether or not a particular neighbourhood is applicable. During the course of the search as constraints are satisfied we expect the signature to change (and consequently allow different neighbourhoods to be bypassed). From the experimental log files we identified the violation signature at every iteration of each run. We then used this

Table 4.2: Results of the one-tailed Wilcoxon Sign Rank test evaluating whether skipping non-applicable neighbourhoods lets a VNS reach significantly better solutions.

	Hard-Soft			Full		
	p value	W	$\bar{x} - E$	p value	W	$\bar{x} - E$
total	0.873	231192.500	-3.500	0.528	228608.000	-1.000
`nhoods1`	0.3	10275.500	-5.500	0.678	6890.500	-0.500
`nhoods2`	0.942	8491.500	-10.500	0.364	10336.500	-5.500
`nhoods3`	0.967	8105.500	-5.500	0.833	8703.000	-3.500
`nhoods4`	0.214	10387.000	-3.500	0.49	9970.500	-6.000
`nhoods5`	0.735	9061.000	-3.500	0.291	9990.000	-1.000

information to calculate how many times the signature changed during each run. Figure 4.4 displays the runs' final number of iterations plotted against the number of signature changes that occurred. The colours represent whether the search could bypass non-applicable neighbourhoods or not. We have also separated out the two partitioning schemes.

For the hard-soft partition it is clear that our assumption about the behaviour of the search did not hold. Irrespective of the number of iterations the search made during the course of a run it did not cause any signature changes. However, for the Full partition runs this is not the case. Those results are more in line with what we had expected; there appears to be a relationship between the number of iterations and the number of signature changes. Runs with more iterations saw more signature changes. Perhaps in retrospect these findings are not so surprising. In the Two-phase COP Partition results discussion in Section 3.6 we noted that there were only four potential signatures. The hard-soft partition runs do find improving solutions, but none ever reach the point of having solved a constraint and therefore they never encounter a signature change.

The Full partition results exhibit a relationship between the iterations and signature changes and also show a difference between the two strategies. The most visually obvious separation is between the `nhoods1` configuration results (shown as the filled squares). The runs where no bypassing occurs (in green) use more iterations than those where bypassing is used (in blue). For the other neighbourhood configurations the plot is not as clear. Table 4.3 shows the results of the one-tailed Wilcoxon Sign Rank test comparing the iterations used by both strategies. We only performed the test on the runs from the Full partition; the plot shows that all the hard-soft runs are the same. The test results confirm that in addition to the `nhoods1` configuration, there are also significant differences between the `nhoods2` and `nhoods4` configurations. These results are strong enough to indicate that regardless of the neighbourhood configuration the iterations are significantly different. We can then conclude that (for the full partition) the neighbourhood bypassing scheme allows the search to reach a local optima using fewer iterations.

Table 4.3: Results of the one-tailed Wilcoxon Sign Rank test evaluating whether the enhanced VNS on the Full partition uses significantly fewer iterations

	p value	W	$\bar{x} - E$
total	< 0.001	390213.500	168.500
`nhoods1`	< 0.001	19900.000	1984.500
`nhoods2`	< 0.05	11431.500	0.500
`nhoods3`	0.917	8732.500	-53.500
`nhoods4`	< 0.001	20059.000	249.000
`nhoods5`	0.383	10091.500	-22.000

We have established that bypassing non-applicable neighbourhoods uses fewer iterations (in a fully partitioned model), but does this also translate into using less time (i.e. does the search reach optima *quicker*)? Table 4.4's results from the Sign Rank test evaluating the time difference are less conclusive. Only for neighbourhood configuration `nhoods5` is there a significant time saving. The number of iterations the search manages are plotted against the time taken in Fig. 4.5. The search procedures are disambiguated by the colours and the symbol shapes (which represent which neighbourhood set was used). The horizontal line on the y-axis is the time-limit. Some of the points are opaque; these denote runs that reached a local optima before the time-limit elapsed. The runs plotted with a solid colour were terminated because the time-limit expired. Out of the 1995 runs, only 9.12 % (182) reached an optima before the time-limit. Fig. 4.5 shows that all of these runs used neighbourhood configuration `nhoods2`. Most of the configurations *time out* slightly before the hard cut-off point. This is because the time limit includes the time taken to parse the problems, build up the constraint model, etc. These slightly reduce the amount of time the search has. Some runs exceed the time limit. This is because the search does not interrupt a neighbourhood exploration that is in progress. Only when the search tries to

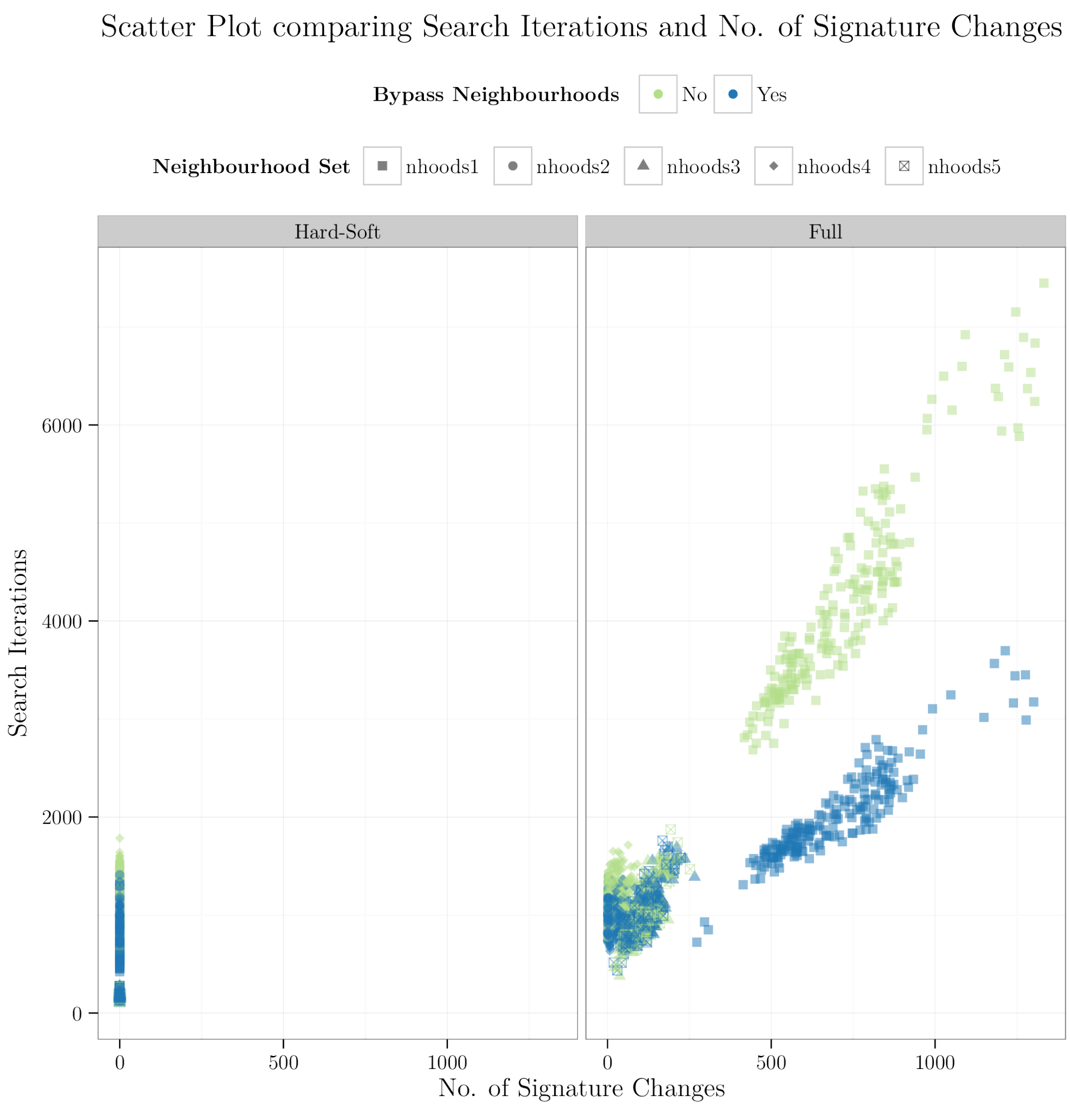


Figure 4.4: Plot of iterations against the number of signature changes

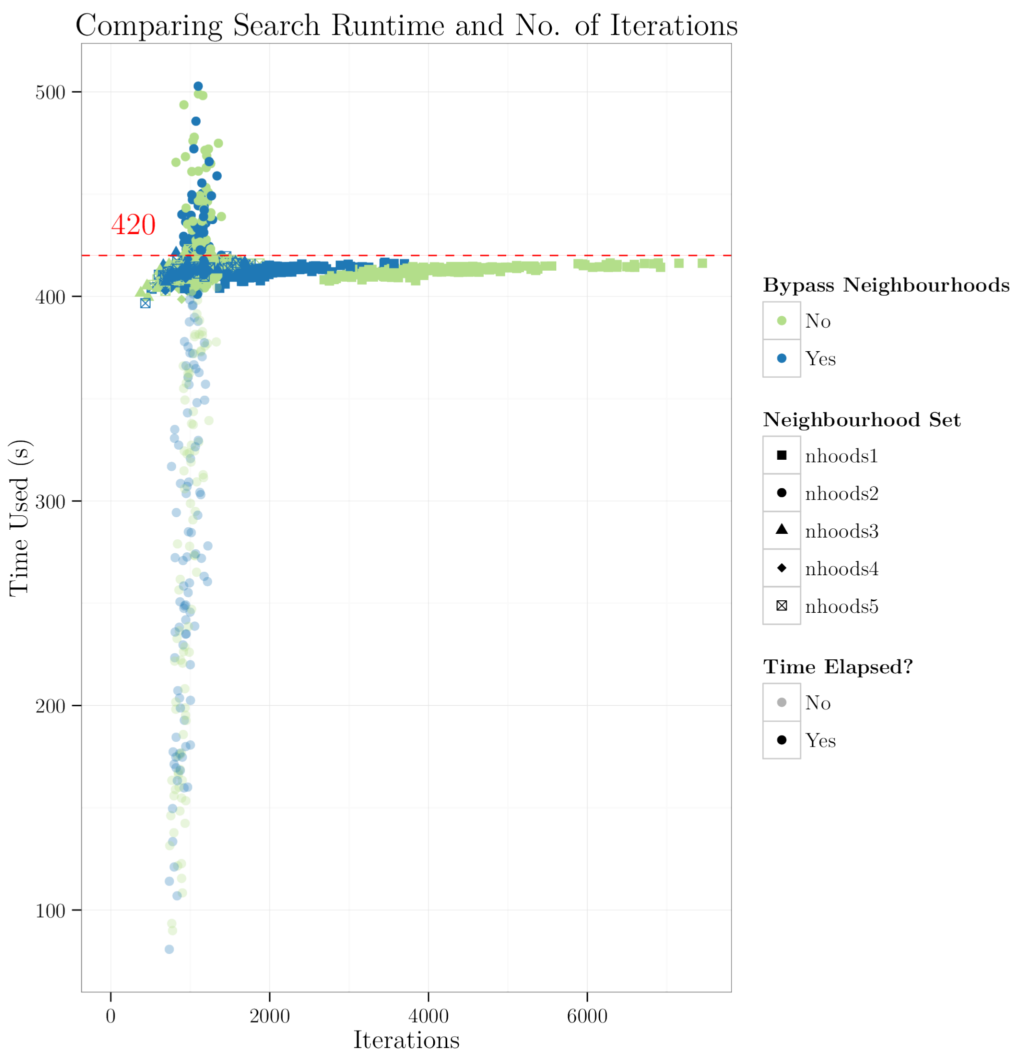


Figure 4.5: Plot showing the time taken (s) against the number of iterations

start the next iteration does it recognise that the time has expired.

Table 4.4: Results of the one-tailed Wilcoxon Sign Rank test evaluating whether the enhanced VNS uses significantly less time

	p value	W	$\bar{x} - E$
total	0.345	251364.000	-132.000
`nhoods1`	0.088	11049.000	-20.000
`nhoods2`	0.562	9923.000	-9764.500
`nhoods3`	0.835	9065.500	-498.000
`nhoods4`	0.686	9652.500	-384.000
`nhoods5`	< 0.05	11221.000	8.000

4.2 Dynamic Neighbourhood Orderings

For VNS / VND it is conventional to have a fixed neighbourhood ordering based on the neighbourhoods' sizes. To order neighbourhoods by size one first needs a way of assessing (or at least upper bounding) the number of moves a neighbourhood will consider. Each class which implements `Neighbourhood<LS>` can be queried for its size. Defining what the size of a neighbourhood is depends upon two factors: the particular problem instance, and the neighbourhood's properties. Some neighbourhoods have a fixed size (for a given problem instance). Consider the neighbourhood `SwapEventTimeslotNeighbourhood`; for it to visit all the swaps there will be potentially $\binom{events}{2}$ moves. Other neighbourhoods have sizes which can vary during the search procedure. For example, the `overlapRemover` neighbourhood selects an event from amongst those scheduled in the same time-slot / room and then attempts to move it to a free time-slot / room space. If there are no overlapping assignments then the neighbourhood will have no moves. Its upper bound occurs when all the events are placed in a single time-slot / room.

The dynamic nature of neighbourhood sizes also raises questions regarding how they should best be represented within the framework. Normally in COMET a value that changes during the course of the search (and is dependent upon other invariants) would be a strong candidate to be an invariant itself. At present the neighbourhood sizes are *not* invariants; they are recalculated by explicitly calling each neighbourhood's `getSize()` method. Invariants can be created and registered with the `Solver<LS>` instance at any point of a COMET program. Once a model is *closed* COMET calculates the update schedule necessary to ensure that invariants are only refreshed when required. However, there is a considerable performance penalty when creating an invariant after the model has been "*closed*". To avoid this, all the neighbourhoods' size invariants would need to be declared at the same time as the problem models. This would introduce a cyclic dependency where a `Neighbourhood<LS>` required a `Model` instance at construction time but the Model could not be fully instantiated until the neighbourhood had been instantiated. By not using a size invariant, we avoided this potential problem and retained a clear separation between models and neighbourhoods (at the cost of some repeated calculations).

Given that neighbourhood sizes can change dynamically, we sought to explore whether there was any benefit to refreshing the VND neighbourhood ordering to reflect the current sizes of each of the neighbourhoods. Non-interacting neighbourhoods *will* still be bypassed. Instead of being a static sequence, the control flow behaviour becomes more akin to Abstract Data Structures such as heaps or priority queues; the first or top element should always be the smallest. This reordering process is also reminiscent of how a CP labelling strategy (such as *first-fail*) will consider the variable with the fewest remaining values for allocation first. The previous section highlighted Curtois et al. [2006] who omitted neighbourhoods based on some expected reward. A similar idea was applied to neighbourhood ordering in Hu and Raidl [2006]. Their Self-Adaptive VND rearranges neighbourhoods based upon previously observed benefits. To reduce the effects of a single particularly good (or bad) exploration, the actual reordering does not happen immediately but waits until the observed benefit passes a threshold.

4.2.1 Experimental Evaluation

The experimental setup was identical to the one described in the previous section (4.1.2). The runs were subject to the same termination conditions and time restrictions.

The results for the twenty ITC instances are displayed in Fig. 4.6. In the figure, some of the same trends from the previous experiment emerge. Again, the fully partitioned model has a clearer delineation between those runs that improve substantially and those that do not. It is also clear that no particular ordering strategy is visually better; the results cluster into groups with a mixture of colours. In the hard-soft partitioned runs the static orderings (in green) consistently appear

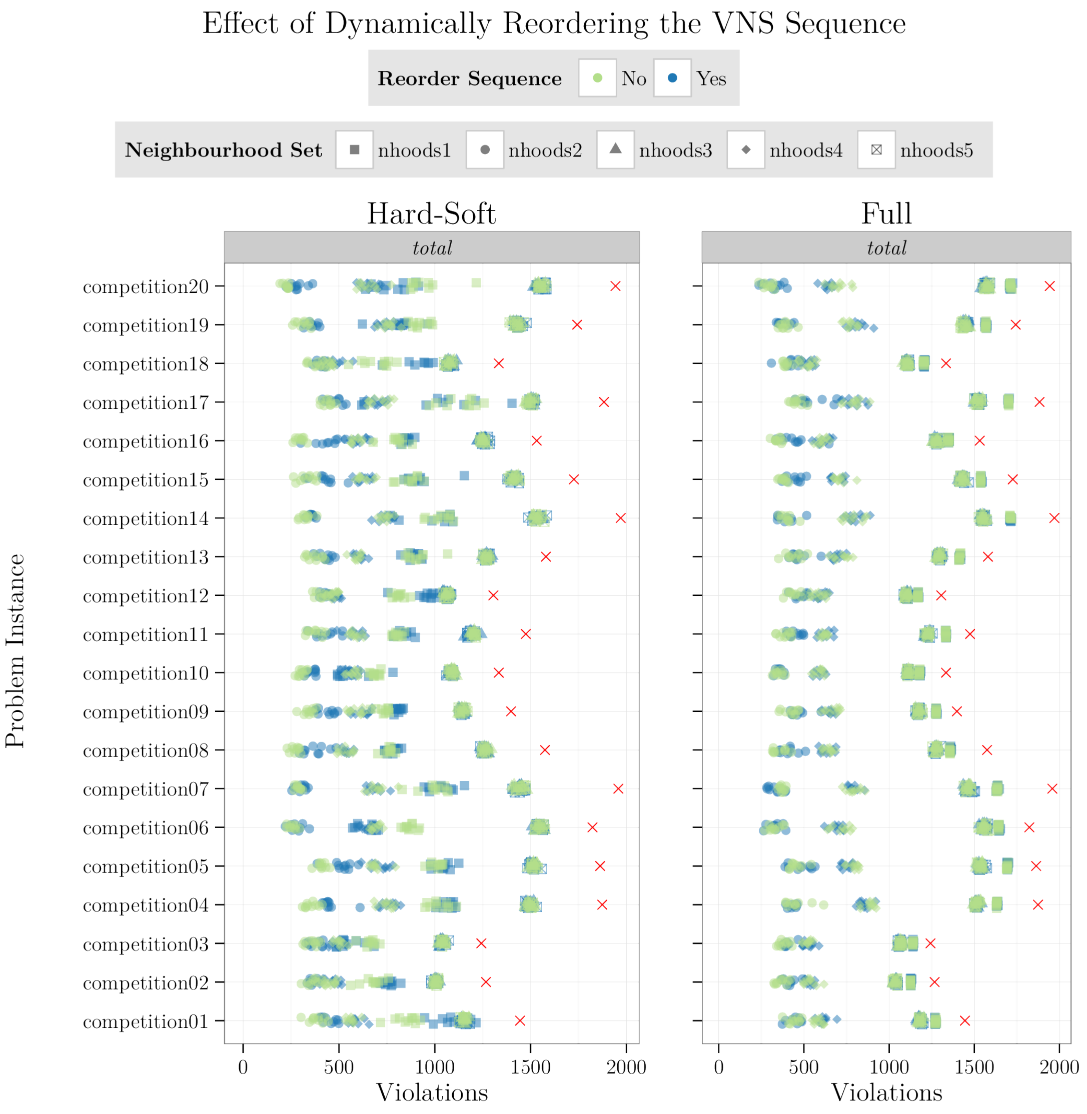


Figure 4.6: Comparing the effect of dynamically reordering the neighbourhood sequence at run-time on the competition instances.

marginally better. This suggests that the reordering is actually slightly detrimental to performance. For the full partitioned runs there appears to be stronger results achieved with neighbourhood reordering, but this only brings them closer to the statically ordered runs rather than giving any distinct advantage. It does not give a clear indication of the benefit gained from reordering neighbourhoods.

Table 4.5: Results of the one-tailed Wilcoxon Sign Rank test evaluating whether dynamically reordering the neighbourhoods offers an improvement over a static ordering.

	Hard-Soft			Full		
	p value	W	$\bar{x} - E$	p value	W	$\bar{x} - E$
total	1	178958.000	-14.500	1	201174.500	-4.000
`nhoods1`	0.842	9136.500	-28.000	1	5014.500	-2.500
`nhoods2`	1	1853.500	-62.000	1	5327.500	-34.000
`nhoods3`	0.545	9177.500	-2.500	0.051	10632.000	-0.000
`nhoods4`	0.871	8664.500	-10.000	0.06	10998.000	-0.500
`nhoods5`	0.579	9496.000	-3.000	0.997	7507.000	-5.500

The results of the Wilcoxon Sign Rank test (in Table 4.5) also highlight that there is no evidence of any significant benefit gained by reordering the neighbourhood sequence. The previous experiment failed to find any advantage to bypassing non-applicable neighbourhoods (in the two partition case) due to the limited number of unique signatures. For an n item list there are n factorial possible orderings, so a lack of diversity is not likely to be the case. However, just because these orderings can exist does not necessarily mean that they will be encountered. A neighbourhood ordering change requires two things: the neighbourhoods need to vary in size, and this variation is large enough (relative to the size of another neighbourhood in the sequence) to result in a change of position.

Figure 4.7 plots the size of the neighbourhoods at each iteration from one of the 2000 runs. This confirms that the neighbourhood sizes do vary during the search; only one neighbourhood from the `nhoods1` configuration had a constant size. The problem seems to be that whilst the neighbourhood sizes do change, the size of these changes are not large enough to affect the relative ordering. There is only one occasion where two neighbourhoods change enough to actually switch position within the sequence. These neighbourhoods are in third and fourth place respectively; the impact of this change will only become relevant if the search fails to find an improvement in (or bypasses) the prior two neighbourhoods.

For each of the 1996 runs that could reorder their neighbourhoods we calculated how often this actually happened. Figure 4.8 displays a table showing the ordering changes broken down by each neighbourhood configuration and model partition. A number of interesting properties are evident. Five of the configurations encounter no situations where the neighbourhood ordering differs from the starting ordering. The runs using the `nhoods2` set have only 5 and 8 changes in the two-phase and full partitions respectively. The remaining plots show most runs experience few ordering changes; out of the `nhoods4` runs 67.17 % saw fewer than 10 ordering changes. One reason we are not seeing any improvement from dynamically reordering the sequences is that the neighbourhoods orderings do not actually change in most cases. Whilst the neighbourhood sizes do change these fluctuations in size are not large enough to result in position changes.

4.3 Constraint Directed VNS

Even with the amendments to VNS seen in the previous two sections the search progress remains primarily driven by the configuration of the neighbourhoods. In this section we propose shifting the focus onto the constraint families rather than the neighbourhoods. We believe that the neighbourhoods available for selection by the search should be dictated by the constraints that remain unsatisfied; we call this Constraint Directed Variable Neighbourhood Search (CDVNS). At any point during the search the violation state of the constraint families is known, and—for any particular constraint family—the constraint-interaction information tells us the potential set of neighbourhoods that could be used. For the Dynamic VNS the search neighbourhood ordering reconfigured itself to reflect the current neighbourhood sizes. In the CDVNS the selected neighbourhoods (and their ordering) will change dynamically depending upon the current constraint violations. No longer will non-interacting neighbourhoods have to be bypassed; they will simply not be selected to start with. Adopting this *constraint-directed* strategy does raise a new issue: what order should the constraints be solved in?

4.3.1 Greedy Orderings

We propose starting with a simple greedy approach: the constraints should be organised in order of increasing violations. The first constraint family that should

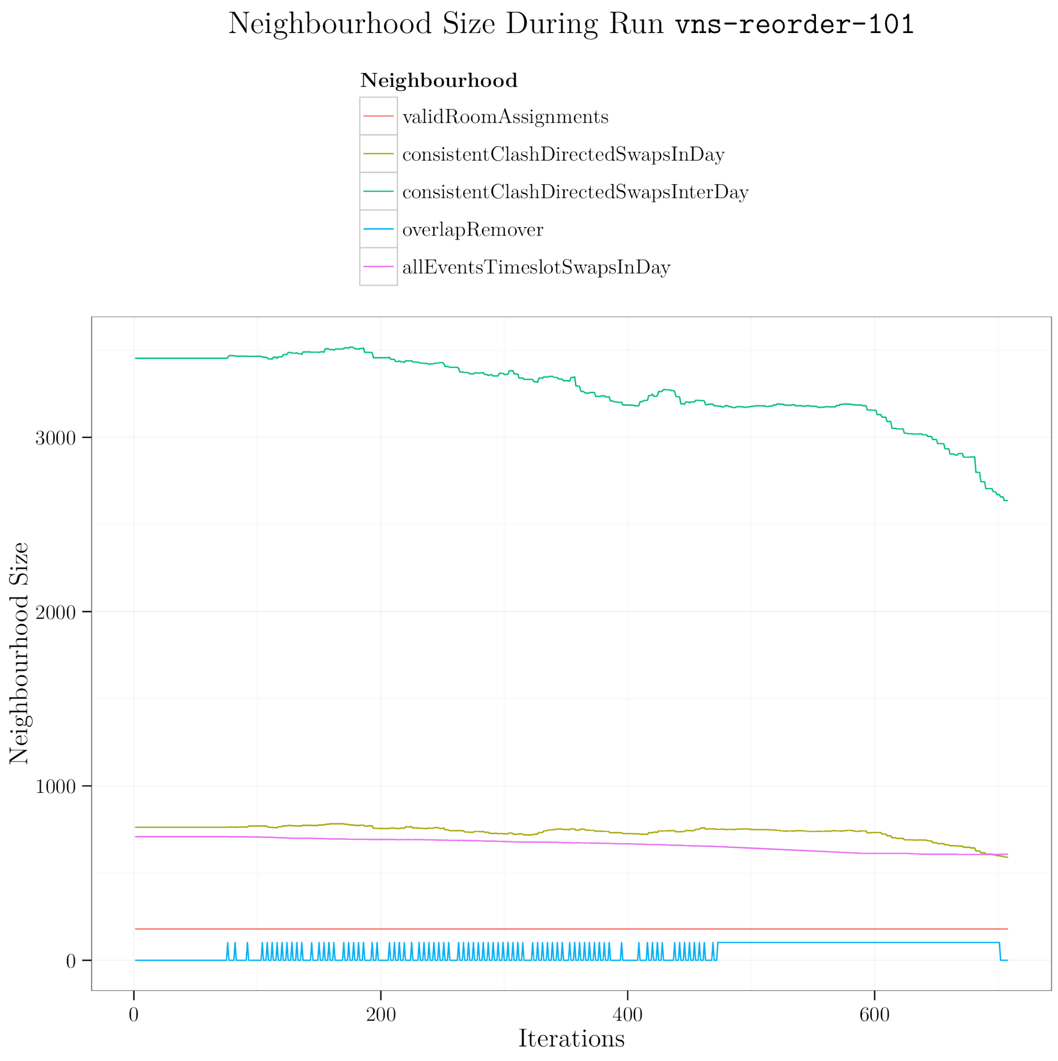


Figure 4.7: Trace of the neighbourhood sizes during execution on instance `competition01`.

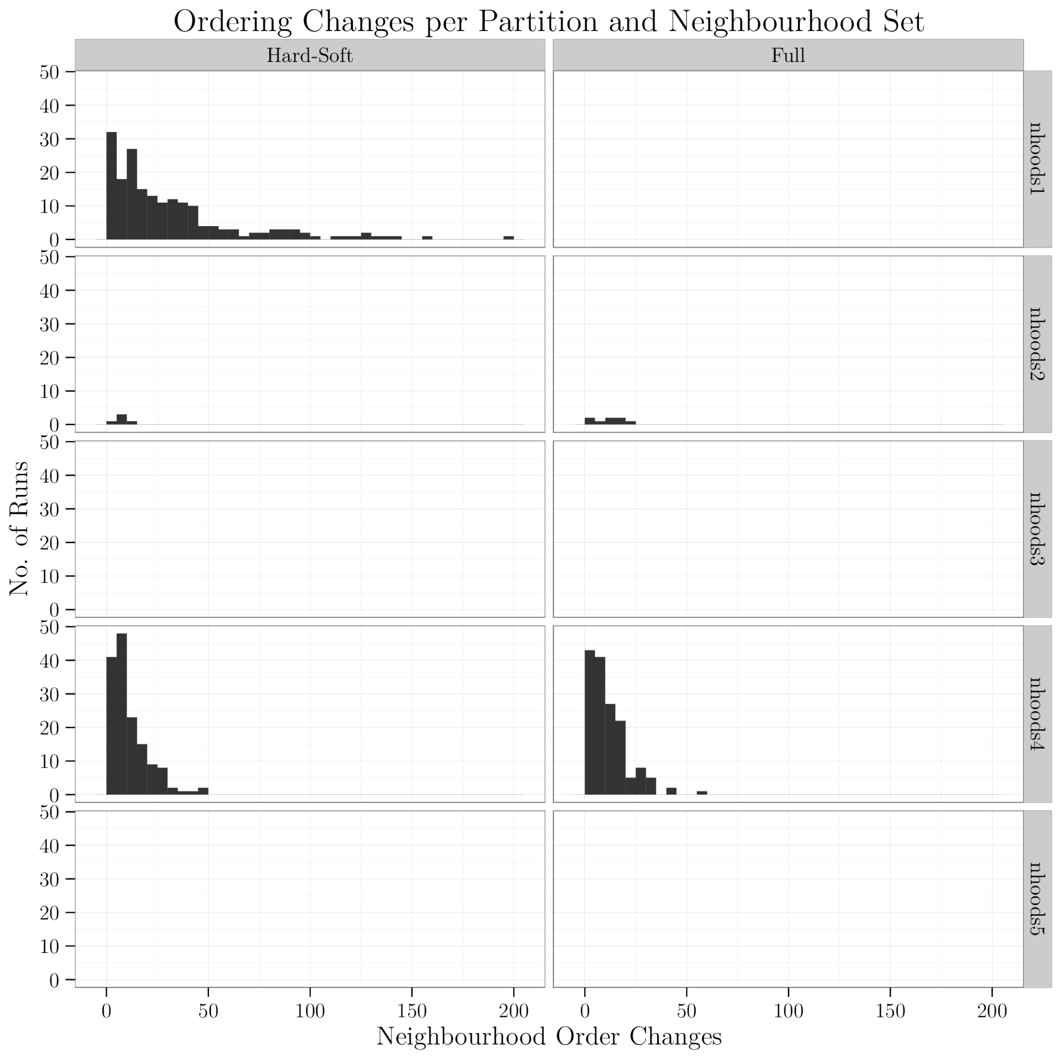


Figure 4.8: The distribution of ordering changes against the model partition and neighbourhood sets.

be used to determine the selected neighbourhoods is the one that is already closest to satisfaction. Once a constraint family is completely satisfied, the search will transition to the next most satisfied family. If, during the course of solving the current selected family, another family appears closer to satisfaction than the selected constraint then it will become the selected family. The intuition behind this style of greedy reactive search is that it will result in a search which focuses intensely on trying to satisfy the constraints that are *almost* satisfied. These families require the fewest improving moves to reach satisfaction, which in turn means less time between signature changes. If violations are reintroduced to previously satisfied families then the search will attempt to rectify these before continuing.

As an alternative to selecting the least violated constraint family, we propose another greedy search that does the opposite. It selects the most violated family. The search behaviour is the same, except that the conditions are reversed; if a family is moved closer to satisfaction than another family the focus switches to the more violated group. If the expectation when selecting the least violated family is a search that focuses on maintaining satisfaction then for this variant we would predict that the search would rapidly improve the commonest constraints. The families with most violations have most scope for improvement. This gradual tightening of violations, bringing the largest family down until it surpasses another family, should lead to a gradual convergence (akin to SA's annealing schedule).

Once again the GLSM formalism provides a concise way of representing the control structure of our algorithm. In this instance, the states represent the constraint families and the transitions occur when there are no violations remaining in a family, or when a previously satisfied family is violated. The implementation of these Greedy Strategies is shown in Listing 4.2. The only difference between the two searches is which invariant set is passed as an argument; one monitors the least violated families, the other the most violated families.

Experimental Evaluation

We compared the performance of the two Greedy variants of CDVNS using the same experimental setup and under the same conditions as the previous experiments. Figure 4.9 displays the results from the twenty ITC instances. The first noticeable trend is that the strategies form clear coloured bands. In both partitionings focusing on the most violated constraints first outperforms concentrating on the

```
void minGreedyOrdering() {
  _violatedConstraints = _leastViolatedConstraints;
  greedyOrdering();
}

void maxGreedyOrdering() {
  _violatedConstraints = _mostViolatedConstraints;
  greedyOrdering();
}

void setupViolatedConstraintChanges() {
  whenever _violatedConstraints@insert(int i) {
    if(!_lookaheadFlag) {
      _shouldStop := true;
      cout << "Pulling the plug on internal VNS" << endl;
    }
  }

  whenever _violatedConstraints@remove(int j) {
    if(!_lookaheadFlag) {
      _shouldStop := true;
      cout << "Pulling the plug on the internal VNS" << endl;
    }
  }
}

void greedyOrdering() {
  int previousNhood = initialNeighbourhood;
  int previousConstraint = initialConstraint;
  setupViolatedConstraintChanges();
  _searchStart = System.getCPUTime();
  while(!_shouldStop && _totalViolations > 0 && hasTimeLeft()) {
    select(c in _violatedConstraints) {
      assert(card(_violatedConstraints) == 1);
      notifyChangeConstraint(previousConstraint, c);
      _scheduler.setValidNeighbourhoods(_cInteractions[c]);
      setNeighbourhoodOrdering(_scheduler.
   getLinearNeighbourhoodOrdering(true));
      int startVNS = System.getCPUTime();
      searchVNS(_neighbourhoodOrdering, false, true);
      cout << "Internal VNS phase finished after " << System.
   getCPUTime() - startVNS << " ms" << endl;
      resetProgressVariables();
      previousConstraint = c;
    }
    onFailure {
      cout << "Failed to find a constraint c in {" <<
   getConstraintNames(_violatedConstraints) << "}" << endl;
      _shouldStop := true;
    }
  }
}
```

Listing 4.2: The Greedy Ordering code from `OriginalSearch.co`.

least violated constraints. The hard-soft results are the stronger than those from the full partition. In the latter partitioning there are few runs that reach below the 1000 violation mark. The hard-soft results are somewhat better with more runs falling between the 500 and 1000 violations range, but when compared to the VNS with bypassing and the dynamically reordered VNS (Figs. 4.3 & 4.6 respectively) they are decidedly inferior. The earlier experiments routinely have runs that have less than 500 violations. The Greedy CDVNS runs have a tighter distribution displaying none of the large gaps evident in the VNS experiments. Another troubling observation is that for many of the runs in the full partition their final solution quality is *worse* than their starting solution.

To investigate the reasons for these poor results we need a more accurate picture of the final optima that the runs reach. Fig. 4.9 displays the *total* violations of each terminal solution, however, each search strategy operates over the violations of the constraint families (visible in a given partitioning). For the hard-soft partitioned runs the search can choose to focus on one of two families; in the full partition the search has six choices. Fig. 4.10 displays just the hard-soft partitioned runs but this time split into the final hard and soft violations. Where the previous figure failed to display any clear difference between the search strategies that is no longer the case in Fig. 4.10. In both the hard constraints and the soft constraints plots the runs form clusters by search focus.

The top hard constraint plot shows that the runs where the most violated constraints were tackled first (shown in green) finish closer to satisfaction than those runs where the least violated constraints were tackled first. All the runs using the max violation strategy improved the hard constraints. The hard constraint results for the minimum violation strategy are not as strong. Some runs do improve from the starting position, but there are others that finish considerably worse. Those using `nhoods4` (the diamond) are consistently the poorest.

For the bottom soft constraints plot the situation is reversed. The minimum violation focus strategy performs best. A divide is apparent between the two strategies with the blue runs almost reaching satisfaction across several problem instances. By contrast the maximum violation strategy improves the soft constraints in only a small number of runs. In the majority of cases the maximum violation strategy ends up introducing more soft constraint violations.

The reason for the stark divide between strategies and the reversal between constraint families becomes apparent when you consider the starting violations. The maximum constraint focus performs best on the hard constraint families because at the starting solution for each instance the hard constraints are the more violated of the two families and consequently are focused on first. Similarly, the minimum constraint strategy performs better on the soft constraints because they start closer to satisfaction and will be selected first.

Figs. 4.11 & 4.12 expand Fig. 4.9 to display the full partition results with each constraint family visible. To keep them readable each plot only shows ten instances. As with Fig. 4.10 the two strategies exhibit clear differences. By exposing the most detailed information about the violations we can see several interesting properties that are not visible in the hard-soft results. Firstly, the initial solutions all satisfy the *overlaps* and *finalTimeslot* constraints. The starting solution generation code and a detailed table of their violation values are in Appendix B. These two families highlight the distinct behaviours of the strategies. The minimum focus approach manages to keep the *overlaps* family satisfied in 811 of the 930 runs; the other 119 runs had only a single *overlaps* violation. Although it is not immediately apparent from the plot the *finalTimeslot* family results are slightly better with 851 runs remaining unviolated. However, the remaining runs are further from satisfaction than with the *overlaps*.

This behaviour is what we would expect to see. These families started with the fewest violations and (as new violations are introduced) become the first that the search attempts to *repair*. The plot shows that the minimum strategy performs well on the *singleEvent* family too. The results for each family become weaker as the number of starting violations increase. By the most violated family, *eventClashes*, the maximum strategy has become the more effective strategy.

4.3.2 Specified Constraint Precedence

The first two configurations of the CDVNS algorithm took a greedy approach to selecting the ordering based on the violation state at run-time. In prior applications of Local Search to constraint solving, this high-level search direction is explicitly made by the algorithm designer. Multi-phase algorithms split the search into distinct sections where the partitioning is explicitly enforced via the neighbourhood selection. The flexibility of our framework using GLSMs as the control structure is that this partitioning is now parameterised. The partitioning decision could be seen as analogous with CP solver's variable (or value) selection heuristics.

Rather than just reacting to the search state, we propose allowing the CDVNS

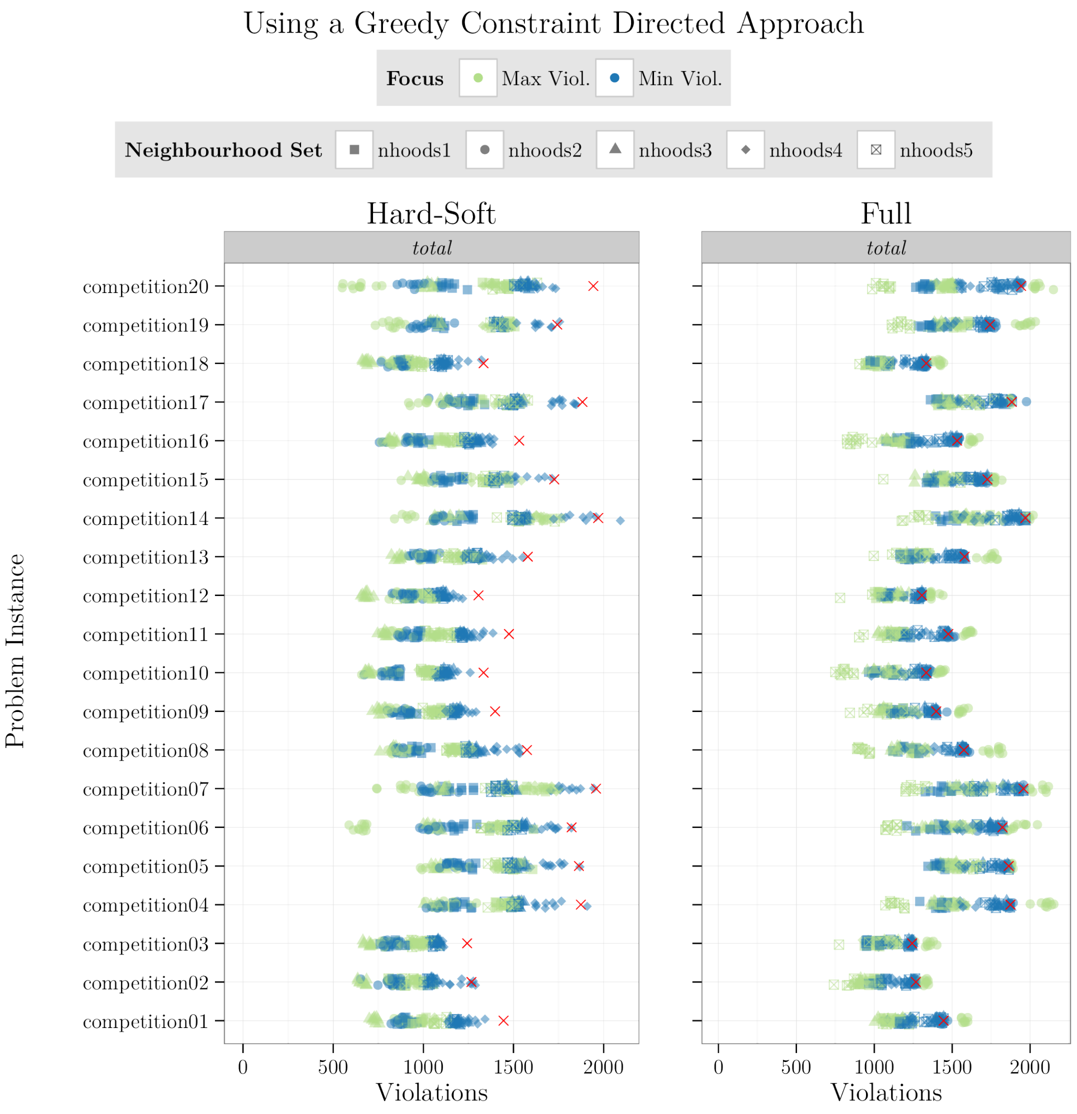


Figure 4.9: Greedy CDVNS focusing on the constraints with the Min and Max Violations.

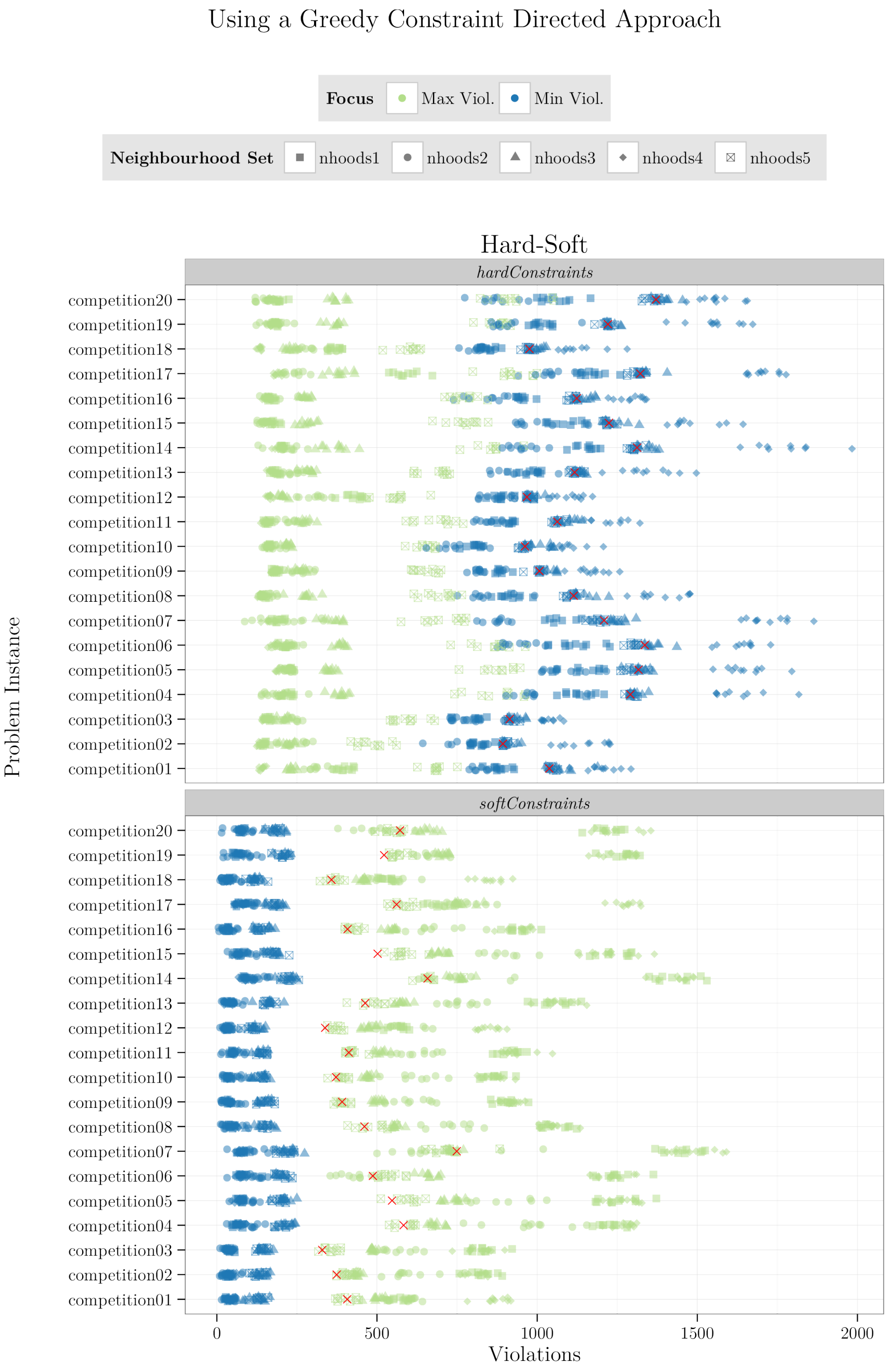


Figure 4.10: Greedy CDVNS results plotted at the constraint family granularity.

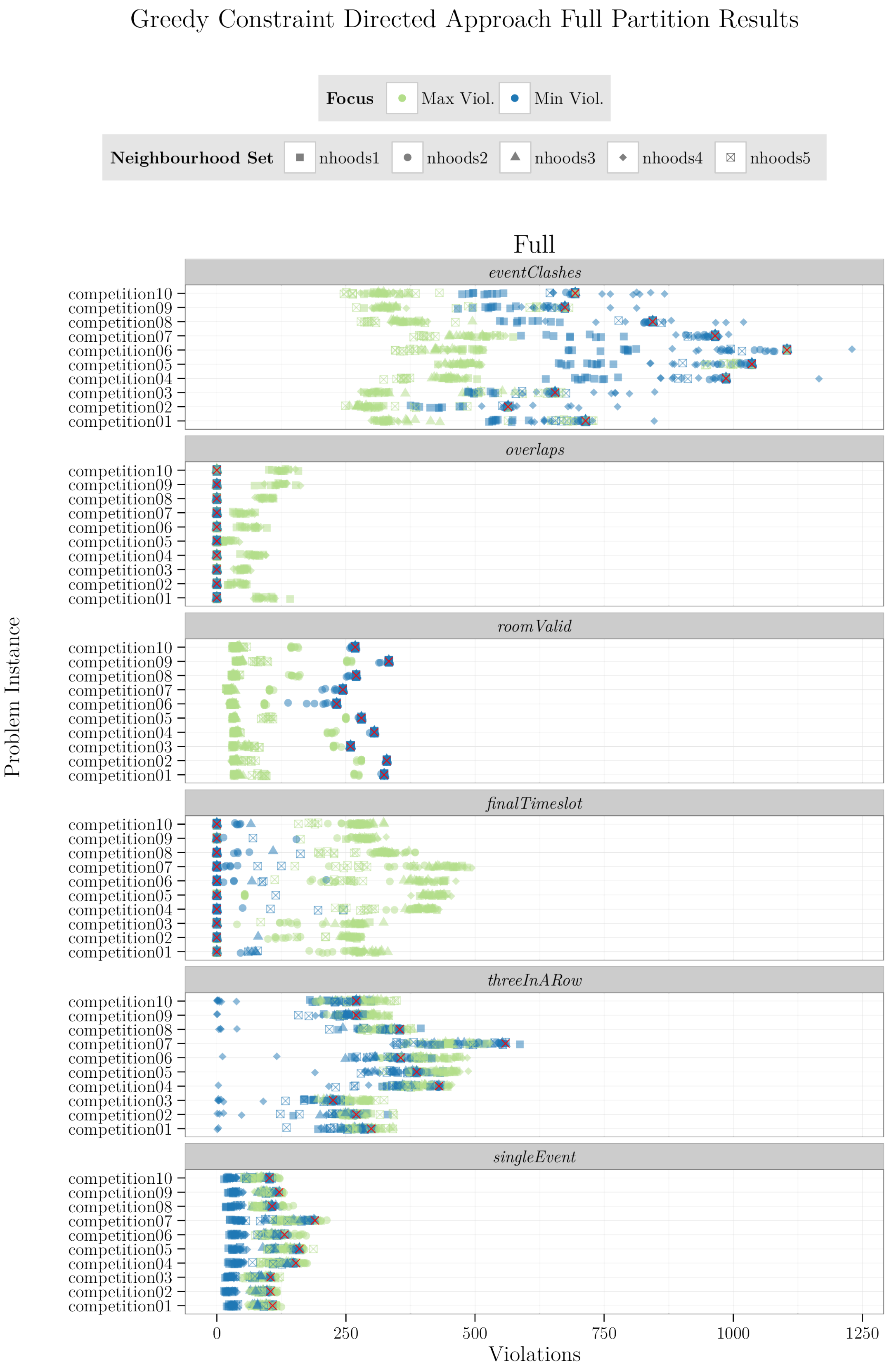


Figure 4.11: Greedy CDVNS full partition results for instances 1–10.

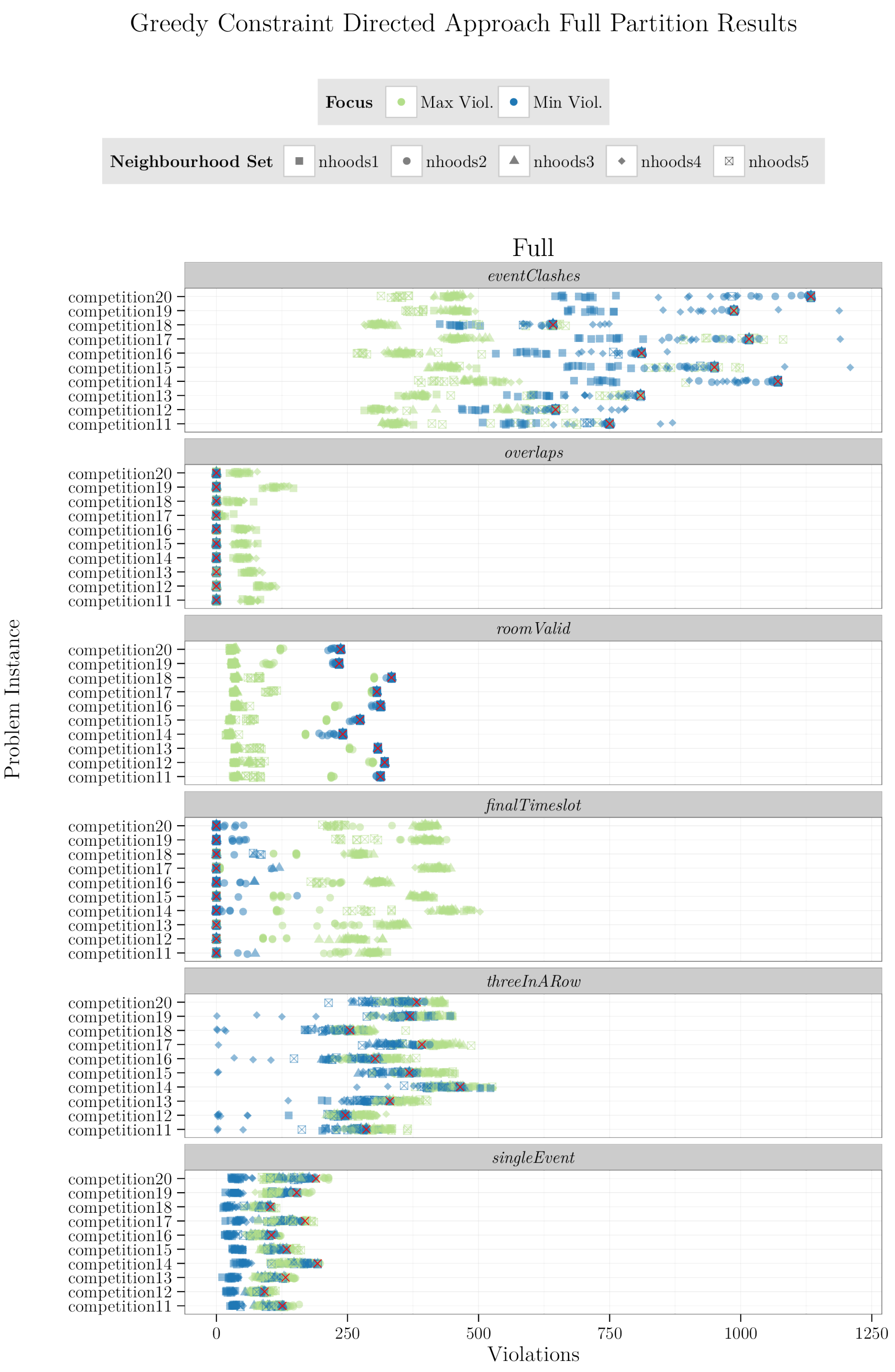


Figure 4.12: Greedy CDVNS full partition results for instances 11–20.

algorithm to be supplied with an ordering of constraints to solve. The search can only transition to focusing on other constraints once it has solved the current set. Introducing violations into previously satisfied constraints requires the search to transition back into those phases.

Listing 4.3 shows the implementation of the CDVNS algorithm. Most of the code handles creating the Event Listeners that COMET uses to detect when the current constraint set has been satisfied (or a previous one violated). The actual search procedure uses the VNS from Listing 4.1. The Constraint-Directed part wraps around the internal VNS and provides meta control. The internal VNS only operates on neighbourhood orderings. It is unaware of the higher-level search decision to concentrate on a particular constraint-family.

CDVNS takes the constraint ordering (defined by the user) then uses that to determine which neighbourhoods are applicable. From those neighbourhoods an ordering is created (using the same method as the VNS). The VNS proceeds as normal, trying to improve the violations of the *selected constraints*. The listeners in the CDVNS procedure are triggered whenever the violated families change and prevent further iterations of the internal VNS (by using a boolean flag). A change in the violated constraints can trigger one of three outcomes: all the constraints from the current partition are satisfied, a subset of the current partition are satisfied, or some previously satisfied constraints are violated.

In the first case the CDVNS search transitions to the next partition (if it has one). The second case does not trigger a transition, but the selected constraint set is updated to contain the remaining violated constraints. The final case causes the search to change the selected constraint set to that of the previously satisfied partition. At each iteration, before starting the VNS, the search refreshes the valid neighbourhoods and the ordering of those neighbourhoods. This means that the applicable neighbourhoods can change during a phase as elements of the selected constraints are satisfied.

Experimental Evaluation

For this experiment we use five distinct constraint orderings. For the hard-soft partition there are only two possible constraint orderings (shown in Figs. 4.13 & 4.14); with six constraint families the full partitioning allows for a wider range of potential orderings (seen in Figs. 4.15, 4.16, & 4.17). For the orderings with groups of equivalent nodes the transitions are shown going to a specific node within those

```
void searchCDVNS(){
   assert(hasConstraintOrdering());
   currentConstraints = getFirstConstraints();
   previousConstraints = new set{int}();

   notifyChangeConstraint(currentConstraints);
   _searchStart = System.getCPUTime();

   whenever _violatedConstraints@remove(int j){
      if(!_lookaheadFlag){
         cout << "_violatedConstraints@remove(" << j << ")" << endl;
         if(currentConstraints.contains(j)){
            notifyChangeConstraint((currentConstraints inter _violatedConstraints)\ {j});
         }

         _shouldStop := true;
      }
   }

   whenever _violatedConstraints@insert(int j){
      if(!_lookaheadFlag){
         cout << "_violatedConstraints@insert(" << j << ")" << endl;
         if(currentConstraints.contains(j) && !selectedConstraints.contains(j)){
            notifyChangeConstraint(currentConstraints inter _violatedConstraints);
         }

         _shouldStop := true;
      }
   }

   while(!_shouldStop && hasTimeLeft()){
      if(!hasSatisfiedCurrentConstraints()){

         _scheduler.setValidNeighbourhoods(getConstraintInteractions(selectedConstraints));
         bool orderBySize = !hasArgument("--ByEffect");
         setNeighbourhoodOrdering(_scheduler.getLinearNeighbourhoodOrdering(orderBySize));
         searchVNS(_neighbourhoodOrdering, false, true);
         cout << "Internal VNS phase finished" << endl;
         resetProgressVariables();
      }else{
         if(hasSuccessorConstraints()){
            advanceToNextConstraints();
         } else {
            cout << "No constraints remaining in the ordering" << endl;
            _shouldStop := true;
         }
      }
   }
}
```

Listing 4.3: The CDVNS code from `OriginalSearch.co`.

clusters. This is a limitation of diagram format rather than our system. The `full1` configuration expresses the same constraint ordering as `hard-soft1` but in a fully partitioned model. Although the constraint ordering is the same, the search behaviour will not necessarily be so. The fully partitioned model allows the search to recognise when some of the families are satisfied (and update its neighbourhood choice accordingly). Configurations `full1` and `full2` are partially ordered. There are precedences between groups of constraints, but within the groups there is no fixed order. The transition condition for these multiple constraint phases are the satisfaction of all the constraints. We use the notation CDET($c_1 \cap c_2$) to indicate that constraints c_1 and c_2 must both be satisfied. We denote the return transitions in a similar fashion using CDET($\neg(c_1 \cup c_2)$).

CDVNS can be seen as akin to what Glover and Laguna [1997, Sec. 10.7, p. 354] call *Referent-Domain Optimization.* Glover and Laguna characterise Referent-Domain Optimization as the process of restructuring a problem (or neighbourhood) with the intention of focusing on a particular heuristic *goal.* This will typically be achieved by creating some restriction on a neighbourhood that either reduces its size or controls the exploration through that space. For CDVNS the problem is being restructured with the intention of solving the constraints in a particular order, which in turn determines which subset of the neighbourhoods will be used.

Other search strategies try to maintain desirable properties through a variety of means (e.g. elite solution pools, GA population, ACO pheromone matrix, TS inclusion memory). CDVNS tries to retain the desirable aspects of a solution (i.e. satisfied constraints) through the use of complementary neighbourhood structures. In Loudni et al. [2010] they outline an interesting hybridisation of VNS, LNS and CP applied to Weighted Constraint Satisfaction Problems. They propose a generic *neighbourhood heuristic* for CSPs that takes into account the topology of the constraints graph (e.g. a graph with nodes for variables and edges where variables appear in constraints). What they term a *neighbourhood* is actually a LNS reassignment of a selection of randomly chosen conflicting variables. By *neighbourhood heuristic* they mean something like the original min-conflicts scheme from Minton et al. [1992]; that is, a strategy for selecting which variables to reassign. The neighbourhoods in their VNS refer to an increasing number of relaxed variables (i.e. those chosen for reassignment) rather than distinct neighbourhood functions. By performing the reassignment with LNS / CP they lose the ability to make predictions about the reassignment (which our work seeks to exploit).

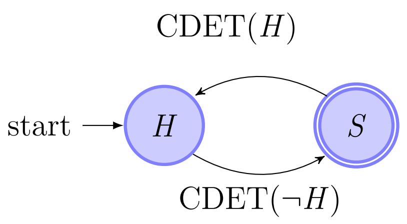


Figure 4.13: The GLSM for a CDVNS using configuration, `hard-soft1.txt`.

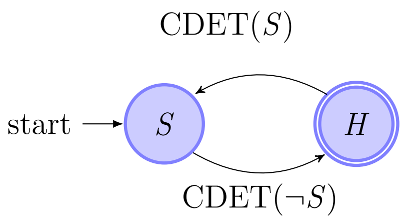


Figure 4.14: The GLSM for a CDVNS using configuration, `hard-soft2.txt`.

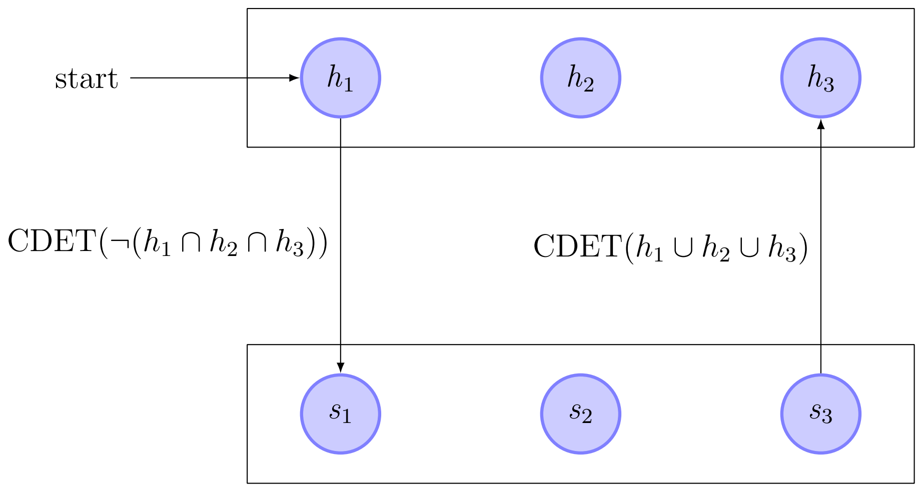

start
h_1
h_2
h_3
CDET($\neg(h_1 \cap h_2 \cap h_3)$)
CDET($h_1 \cup h_2 \cup h_3$)
s_1
s_2
s_3

Figure 4.15: The GLSM for a CDVNS using configuration, `full1.txt`.

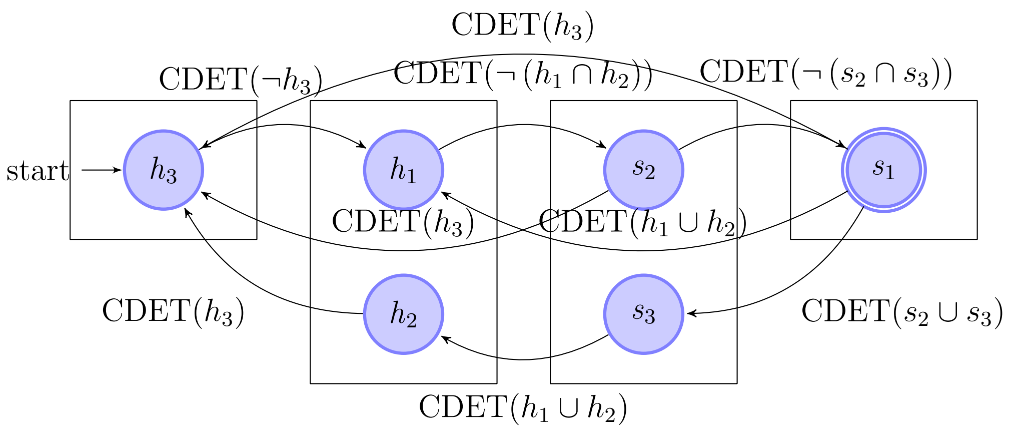


Figure 4.16: The GLSM for a CDVNS using configuration, `full2.txt`.

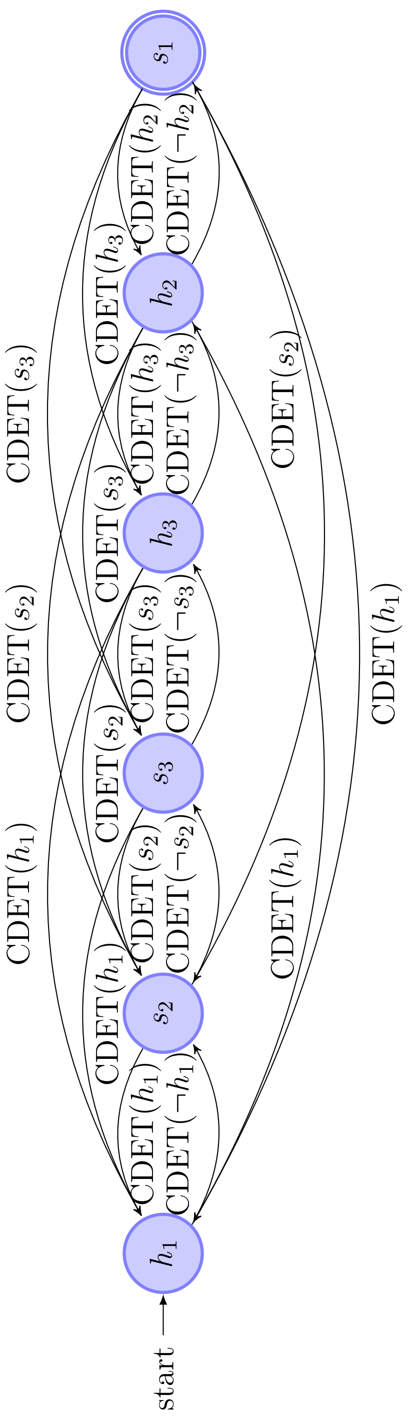


Figure 4.17: The GLSM for a CDVNS using configuration, `full3.txt`.

The ordered CDVNS experiment used the same setup as the prior experiments. The results are displayed in Fig. 4.18 and upon initial inspection do not appear substantially different to those from Fig. 4.9. The hard-soft runs are slightly stronger than those from the full partitioning, but no one ordering appears clearly better than any other. As with the two greedy strategies in the fully partitioned experiment some of the runs are actually worse than the starting point.

Expanding out the hard-soft partition results (in Fig. 4.19) gives a similar picture to Fig. 4.10; the differences between the two orderings (`hard-soft1` & `hard-soft2` exhibit the same trend as the difference between the two greedy strategies. Each performed well on only one of the constraint families. In this experiment the orderings were dictated by a predefined user input, but the outcome was actually the same as that generated by the greedy strategy. In the `hard-soft1` ordering the search tried to satisfy the *hardConstraints* family first and would only focus on the *softConstraints* once the former were feasible. The maximum focused greedy strategy also tried to solve the *hardConstraints* first, but because they were the more violated constraints rather the due to any external prompting. Though the initial impetus was different, the focus, behaviour and results ended up the same.

Whilst the hard-soft orderings ended up mirroring the behaviour of the greedy strategies, the full partition orderings should be capable of more nuanced search behaviour. The detailed full partition results are shown in Figs. 4.20 & 4.21. Again the most noticeable factor is that no single ordering emerges as substantially better than the others. The `full2` ordering maintains the *overlaps* feasibility well, because this is first family in its ordering (see Fig. 4.16). Compared with Figs. 4.11 & 4.12 the user-directed search is also more effective on the *eventClashes* constraints. Again this family occurs is amongst the first constraints tackled in all three orderings. Technically `full2` will start with the *overlaps*, but these will be satisfied in the initial solution, so should be skipped immediately. The performance on the *threeInARow* constraint is worse than in the greedy strategies. The majority of runs end up with more of these violations than they started with. The *finalTimeslot* results are the same, although they start from feasibility so we would expect some violations to be introduced. The *finalTimeslot* family are always last in the three orderings, so any violations that are introduced by neighbourhoods used in other phases are never a priority.

The other factor contributing to some runs' poor performance is that the search acceptance is based solely on the *selected constraints*. A move will be accepted if it

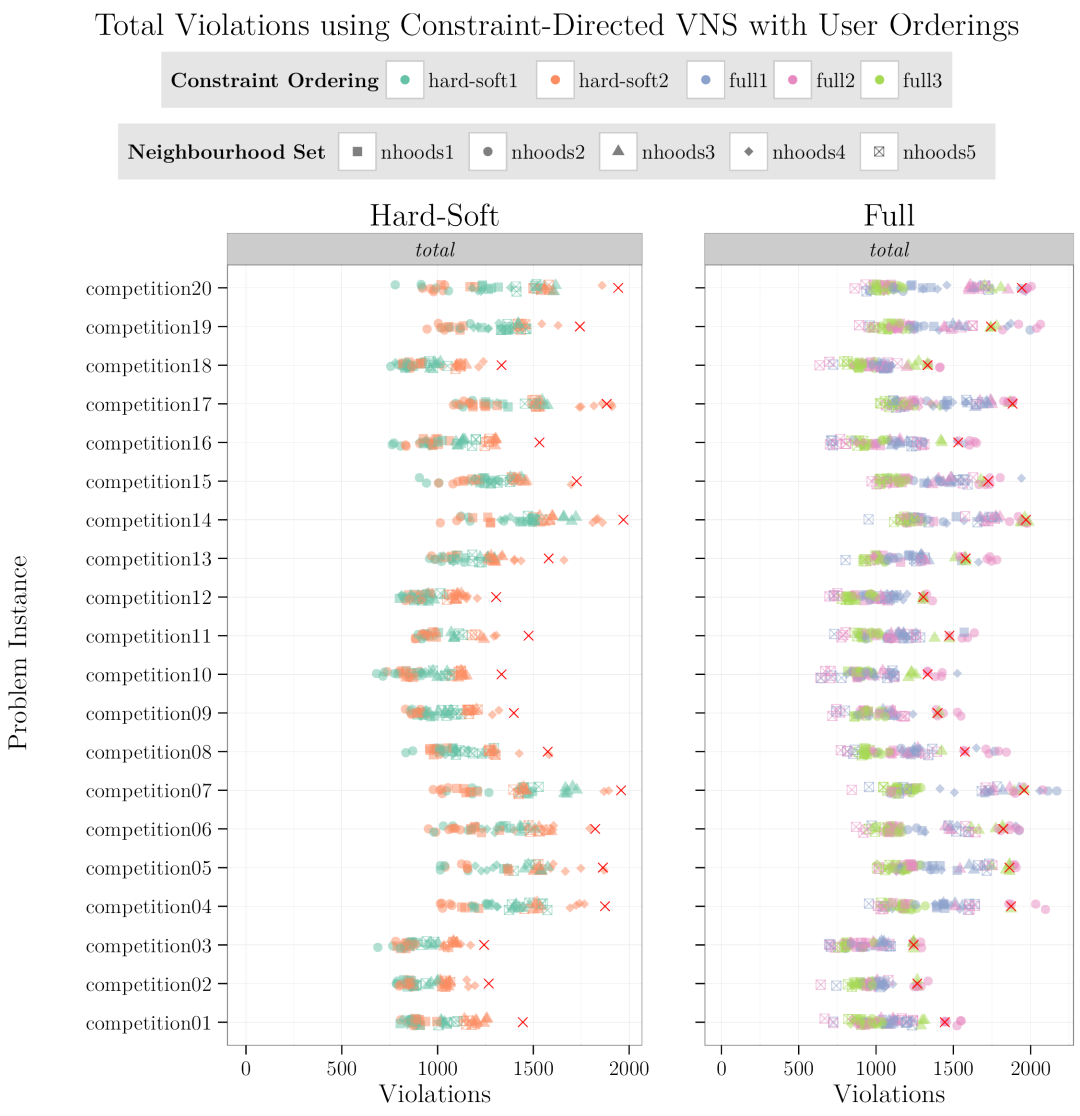


Figure 4.18: CDVNS following user constraint orderings.

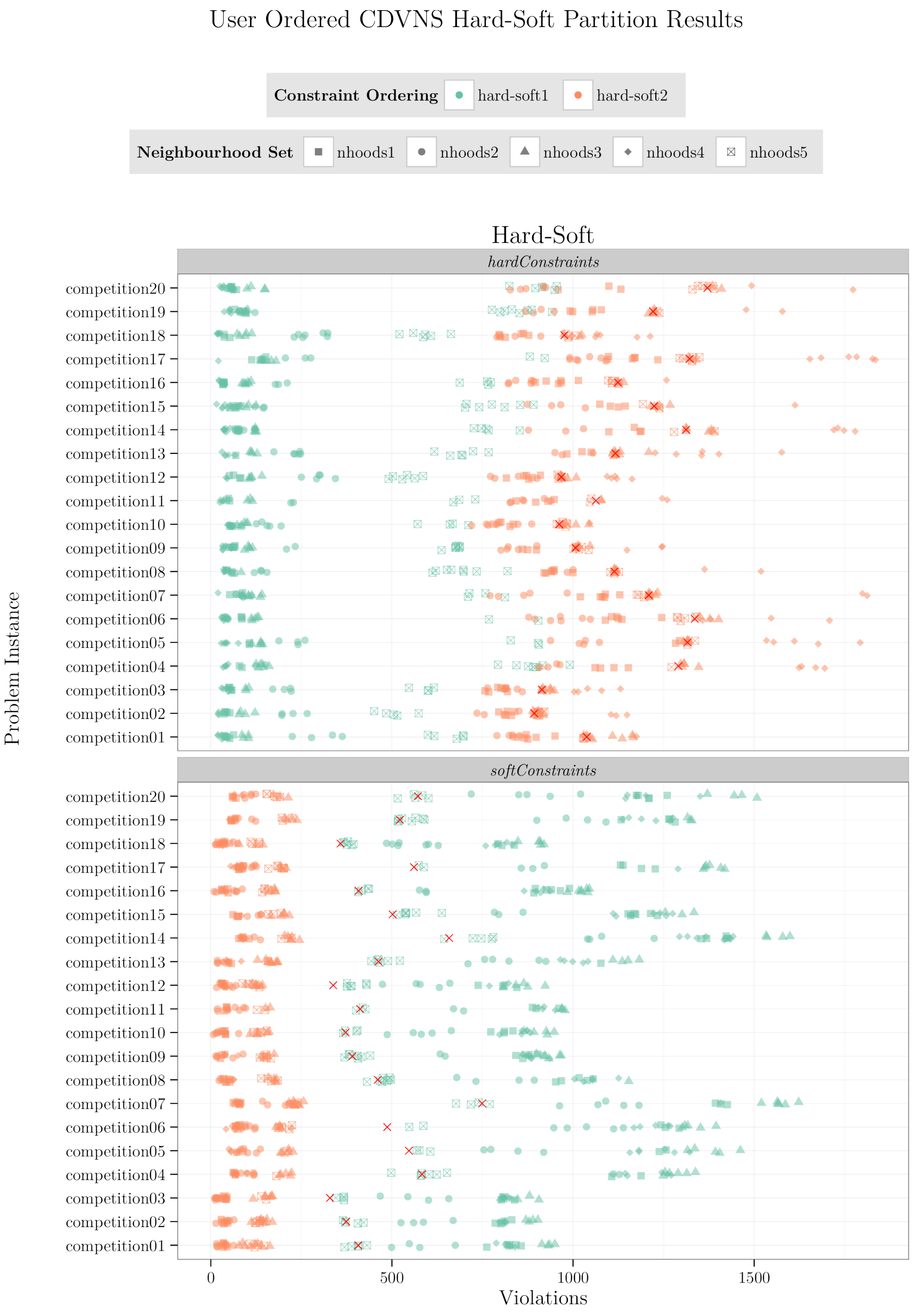


Figure 4.19: CDVNS ordering displaying just the hard-soft partition.

improves (or does not degrade) the violations of the constraints within the selected constraints set. This can, and does, lead to situations where whilst the desired aspect is improved the constraints outside the search's immediate attention are violated more. We experimented with applying a further restriction that a move would only be accepted if it did not degrade the selected constraints *and* it did not increase the summation of the non-selected constraints. So, the other constraints' violations could alter (and increase) as long as the overall effect was neutral. This did not lead to any substantial alteration in the search performance. The main difference was that the search would become trapped at optima earlier.

4.4 Providing Modelling Feedback

The final potential use for constraint-neighbourhood interaction information that we present is to allow feedback to be given to the user regarding their choice of model or selection of neighbourhoods. There exist numerous static code analysis tools for mainstream languages which can provide a programmer with information on best practises or spot potential coding errors. The PMD project[1] for Java allows source files to be scanned and makes warnings based on a flexible rule-set of potential bugs. Most compilers will produce warnings informing the user that there are issues like unused variables or unsafe type casts. Modelling problems well as CSPs requires experience and an understanding of the relationship between the model and the search. In Van Hentenryck and Michel [2005, Chp. 4, p. 45] they introduce a model for the Magic Square Problem which is missing one constraint stating that all the values must be different. They acknowledge this omission but because their construction procedure maintains the *all different* property and the subsequent search neighbourhood is a *2-opt* swap scheme they reason the explicit constraint would have been redundant. This is an example of where a constraint model has been simplified by understanding the behaviour of the search procedure against that model. An inexperienced user may not necessarily be aware of this sort of optimisation. By reducing constraints the user can save both memory and time (certainly in the initial instantiation stage).

Detecting redundant constraints could be automated given only the constraint-neighbourhoods interactions graph (and the list of selected neighbourhoods). It is essentially analogous to detecting the connected components in a graph. Any

[1]`http://pmd.sourceforge.net/`

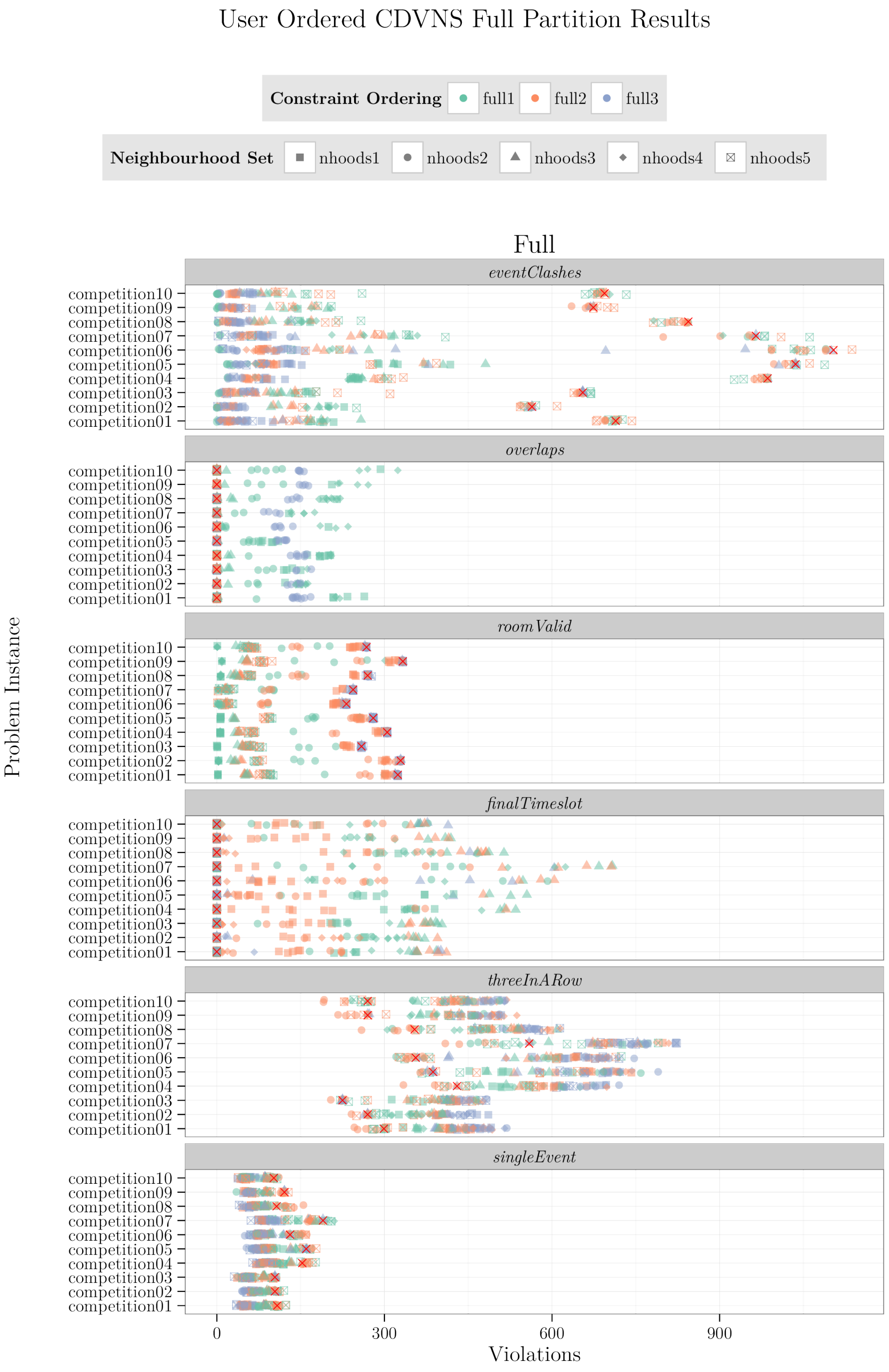


Figure 4.20: CDVNS full partition results on instances 1–10.

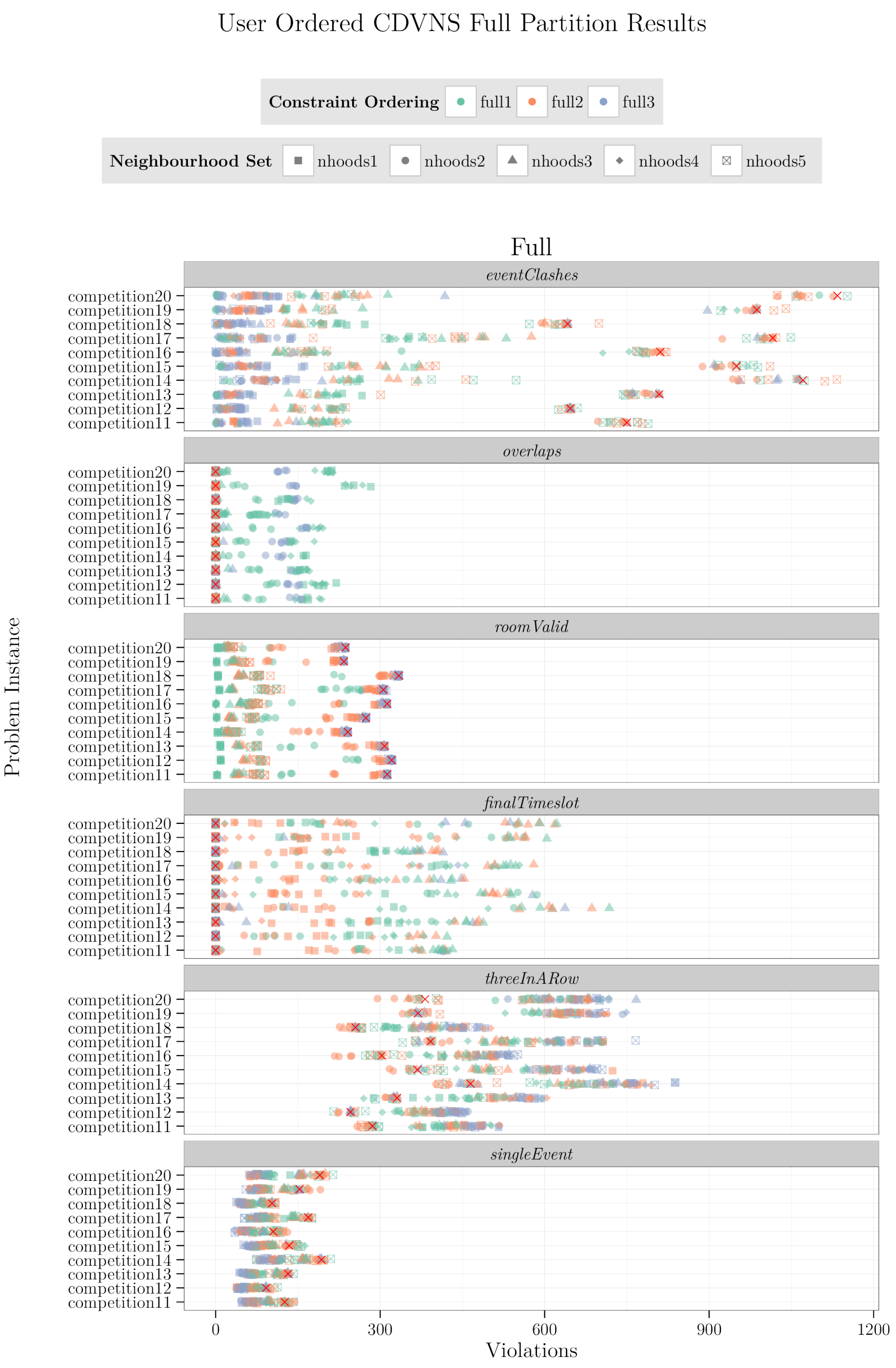


Figure 4.21: CDVNS full partition results on instances 11–20.

of the tree-searches from Section 2.3.1 would work. Disconnected vertices are redundant and can be flagged to the user. Providing modelling feedback may not seem as important a contribution as improving an algorithm's performance, but it does play a role in increasing the accessibility of CBLS. The easier-to-use and more user-friendly a system is, the more likely it is to achieve widespread adoption.

4.5 Conclusions

In this chapter we have explored several ways that constraint-neighbourhood interaction information can be incorporated into Local Search algorithms. We have suggested improvements to the uninformed VNS / VND strategies that allow them to harness information about neighbourhood behaviours. We have investigated whether restructuring the neighbourhood sequence to reflect the changes to neighbourhood sizes is beneficial. We presented a new algorithm, CDVNS, that behaves like a VNS which orders neighbourhoods according to the constraint situation. We experimented with two forms of this algorithm: one that greedily reacts to the current violation state, and one that is guided by a user ordering of constraints.

What emerged from these experiments was that none of our proposed uses of the interaction information we had extracted in Chapter 3 provided any statistically significant improvement over not using any information. The initial premise that using interaction information as a means of avoiding redundant exploration during a VNS seemed plausible; however, what transpired was that the violation state of the problem was too coarse and changed less than we had expected. This, in turn, restricted the opportunities for altering the neighbourhood ordering and each run effectively had a fixed ordering (though not necessarily the same as the full sequence).

Our second experiment—allowing the neighbourhood ordering to change at each iteration to reflect the current neighbourhood sizes—also failed to demonstrate any significant improvement. Upon further investigation we discovered that whilst the neighbourhoods did alter in size these changes were not large enough to trigger a change in the overall order. In the situations where the sequence order did change the differences were between later neighbourhoods that were only accessed infrequently.

The third set of experiments looked at using the constraints' violation state to derive a neighbourhood ordering (rather than having some predefined notion of a neighbourhood sequence). Our first two approaches were greedy and chose to focus on either the most or least violated constraint family. These produce results where clearer behaviours emerged than in prior experiments. Upon investigating the results at the constraint family level of granularity what became apparent was that the division between the strategies was chiefly influenced by the starting solution. The minimum violations approach arrived at solutions where those constraints that started with fewest violations were improved most. Conversely, the maximum violations approach did the exact opposite. The families that started with most violations were improved whilst those families initially nearest satisfaction were degraded.

The last experiments continued with the idea of the neighbourhood ordering resulting from a constraint ordering but, instead of being dynamic, this constraint ordering should be specified by the user. For the hard-soft runs the limited constraint orderings available meant that these were unintentionally the same as the earlier greedy strategies (and exhibited the same results). For the full partition runs richer constraint orderings were possible, but none of them emerge as substantially better. The families that the user-directed CDVNS performs best on are those that appear earlier in the constraint ordering (and also where it has an effective neighbourhood for that type of violation).

In the next chapter we summarise all the work from this—and the preceding—chapter, and evaluate whether we achieved the objectives set out in the introduction. It also explores some of the directions in which we envisage our work being extended and continued.

Chapter 5

Conclusions and Future Work

There is time enough for everything, in the course of the day, if you do but one thing at once; but there is not time enough in the year, if you will do two things at a time

Letters to His Son, April 14, 1747
Philip Stanhope, 4th Earl of Chesterfield

The previous two chapters contained all the experimental work of this thesis. Chapter 3 was concerned with the identification of constraint-neighbourhood interactions. The subsequent chapter explored how the constraint-neighbourhood interaction information could be applied in search procedures. This chapter looks at whether the work in Chapters 3 and 4 met the goals set out in Section 1.1 of the Introduction. It reviews each of the targets, recaps on the relevant work, and discusses whether it satisfied the goal. The first section of this chapter focuses on the constraint-neighbourhood interaction detection segment of the work. The following section looks at the utilisation of the constraint-neighbourhood interactions. The final section provides some closing remarks about our work and its intended aims.

The hypothesis that we set out was that *relationships between generic constraint models and search neighbourhoods can be detected automatically and subsequently used to inform a search procedure.*

5.1 Conclusions

5.1.1 Extracting Interaction Information

The central question motivating the work in Chapter 3 was: *can useful information be found using a generic model and search neighbourhoods*? The first step towards answering this was to investigate how a model and neighbourhood could be related to each other. Specifically, we sought to determine the following: *How can the CSP problem definition be connected with the search behaviour*?

We proposed the concept of *constraint-families*, which are groupings of the constraints within a CSP. The notion of constraint-families allowed us to produce a concise definition of what we termed a *constraint-neighbourhood interaction*. Any neighbourhood that can alter the violation state of a family of constraints is said to have an *interaction* with that family. The families that a neighbourhood interacts with are termed its *constraint signature*.

The second question was: *will this require any modelling changes*? The conventional modelling practice in CBLS, inherited from CP, is to have all the constraints in a single constraint store (e.g. a COMET `ConstraintSystem<LS>`). Maintaining multiple constraint families requires moving to an arrangement where the constraints in each family are collected in independent `ConstraintSystem<LS>`s. We investigated whether this would cause any appreciable time or memory overheads. The results at the end of Section 3.1 show that the time and memory increases were minimal. Content that implementing constraint families was technically feasible, we also investigated some other potential benefits. We found that using constraint stores other than `ConstraintSystem<LS>` could be more efficient; they can also provide access to a wider range of differentiation methods (which allow for more expressive neighbourhoods).

COMET provides useful features—like differentiable invariants, **neighbor**s, and continuations—that make developing CBLS algorithms concise. Van Hentenryck and Michel [2007] extended COMET with a generic **model** system that allowed a generic constraint model to be treated as a first-class expression. Our third question asked: *What kind of framework is needed to create a generic reusable system*? The **model** is limited to only two constraint families: hard constraints, and soft constraints; Van Hentenryck and Michel's **model** cannot represent arbitrary partitionings and as a result we had to create our own generic `Model` architecture. The **model** syntactic shortcuts do show how this could be achieved neatly. Section 3.3 outlines our design for a framework built atop COMET that includes a Model section and new Neighbourhood architecture. As of version 2.1.1 of COMET, neighbourhoods remain coupled to their respective models; this dependency is caused by the need for a reference to the constraint store being differentiated to appear in the neighbourhood. Our framework addresses this problem by allowing neighbourhoods to be completely disjoint from the models they will be used with. However, this separation is not mandatory; it is still possible to create neighbourhoods that rely on a specific model (e.g. see the timetabling neighbourhoods described in Appendix A.2.2).

The role of the framework is to create the environment where an independent analysis component can operate. The last subquestion we sought to answer was, given decoupled models and neighbourhoods, *how can the behaviour detection be automated*? The Interaction Detector accepts a model and a set of neighbourhoods as arguments and can then uncover the constraint-neighbourhood interactions that exist between them. The detector employs a simple strategy whereby it starts from a random solution, makes random moves within a neighbourhood, and uses an Event Listener to capture any situation when the violation state changes. To verify its efficacy we performed a number of experiments using the Timetabling Problem as a test bed. The results found in Section 3.6 confirm that this policy does perform well. The detector was able to uncover all the interactions for all neighbourhoods across three different constraint partitionings.

Having reviewed the initial information extraction goals it would seem reasonable to conclude that the work in Chapter 3 did fulfil them. Whilst the chapter does describe a generic way of extracting information from models and search neighbourhoods, it does not prove that this information is *useful*. To resolve the utility of the interaction information required a study of potential applications during the search process.

5.1.2 Exploiting Interaction Information

Chapter 4 contains the second half of the work in this thesis. It starts from the position that for a given constraint model and set of neighbourhoods we have access to the constraint-neighbourhood interactions information. The main question that this chapter sought to answer was: *how can this [constraint-neighbourhood interaction] information be used within a search procedure with the aim of improving performance*?

As part of that question we started by looking at how neighbourhoods are used within an existing VNS context. At present neighbourhoods have to be explored and found to contain no improving solutions before the search can transition to the next neighbourhood in the VND sequence. Given that interaction information can indicate situations where a neighbourhood will never be able to alter the violation state we experimented with using this knowledge to bypass non-applicable neighbourhoods. The hypothesis was that avoiding provable redundant exploration would increase performance, however, we failed to find any strong support for this. Further investigation showed that was because what we termed the violation signature (i.e. those constraint families that remained unsatisfied) did not change often. In the simpler two family partition there were no situations where the signature changed, so the same neighbourhoods remained applicable and no bypassing opportunities arose. In the richer full partition model we did encounter signature changes and more bypasses occurred. This did not lead to higher quality solutions, instead we discovered the search reached equivalent local optima but in fewer iterations.

As part of the structure of our system we chose to represent the sequence of neighbourhoods to explore as a GLSM. Rather than being fixed the order of neighbourhoods were simply transitions within a state machine that could be altered dynamically at run-time. This configuration allowed us to experiment with reordering the sequence of neighbourhoods during the search to keep it sorted linearly by increasing size. Maintaining the neighbourhood ordering in a strict size failed to give any significant performance benefits. Whilst the neighbourhoods were displaying sizes that varied during the course of a search these changes were not large enough in most cases to cause a change in the sequence order.

Rather than manipulating a linear ordering of neighbourhoods, we explored allowing the neighbourhoods to be selected based upon the violation signature of the search at that point. Only neighbourhoods applicable to the current violation state would be chosen (removing the need to bypass non-applicable neighbourhoods). We tried two different policies for determining which neighbourhoods should be selected: one guided by a greedy rule, the other following user direction. The greedy approaches selected neighbourhoods by first selecting either the most (or least) violated constraint families and then using only their interacting neighbourhoods. What emerged from these strategies is that resulting optima were likely to be determined by the relative starting violations. This was particularly evident in the hard-soft partitioning. For the user constraint orderings the simpler hard-soft pair essentially replicated the effect of the greedy rules. The full partition showed a more nuanced behaviour. Constraint families could be prioritised through the user decision about their position with the ordering. None of the user-specified orderings generated great results.

Having recapped the work in Chapter 4 overall we think we only partially met our objective of proving the *relationships between generic constraint models and search neighbourhoods can be detected automatically and subsequently used to inform a search procedure.* The first element around formalising the relationships and creating a way of automatically detecting them *was* successful. We created a definition of one potential relationship between constraint models and neighbourhoods (in the form of constraint-neighbourhood interactions). It should be noted (as we set out in the Caveats in Section 1.1) the intention was *not* to claim that constraint-neighbourhood interactions are the only relationship—only one example of such. Our other main contribution was the Interaction Detector (and the work establishing the modelling changes and framework to support it) which showed that uncovering these relationship automatically (and accurately) was feasible. Again though the intention was to prove its feasibility and we offer no claims that the Interaction Detector as it stands is necessarily the most efficient way of uncovering such connections. For the second section of our hypothesis we put forward several ways of using the interaction information with a search. None of the methods we experimented with resulted in great performance, so in that respect this aspect of the work was not successful.

5.2 Future Work

The remainder of this chapter covers some of the directions in which our work could be extended in the future. Neighbourhoods—and their role within Local Search—have been central to this thesis; however, they remain ill-defined structures primarily shaped by prior experience. The next two subsections look at potential strategies to rectify this. Section 5.2.1 presents some ways that the neighbourhood design process could be automated (or at least vaguely formalised via guidelines). In the earlier chapters we focused on the connections between neighbourhoods and constraints; Section 5.2.2 proposes investigating the relationships between the neighbourhoods themselves. The third subsection returns to the Interaction Detector from Chapter 3 and looks at how it could be made more efficient. The final subsection outlines another potential use of the constraint-neighbourhood interaction information and the GLSM structures: the automatic inference of multi-phase algorithm structures.

5.2.1 Neighbourhood Design

In the following quote from Papadimitriou and Steiglitz [1998, Chp. 19, p. 455] they set out what they consider to be the fundamental decisions when adopting a Local Search approach:

> ... [W]e must choose a "good" neighborhood for the problem at hand, and a method for searching it. This choice is usually guided by intuition, because very little theory is available as a guide. One can see a clear trade-off here, however, between small and large neighborhoods. A larger neighborhood would seem to hold promise of providing better local optima but will take longer to search, so we may expect fewer of them can be found in a fixed amount of computer time. Do we generate fewer "stronger" local optima or more "weaker" ones?
>
> These and similar questions are usually answered empirically, and the design of effective local search algorithms has been, and remains, very much an art.

The size and quality of the optima are important in neighbourhood design; so too is a neighbourhood's interaction signature. The ideal neighbourhood—for the purposes of our work—is one that interacts with as few constraint families as possible. A neighbourhood with few interactions will behave in a more predictable fashion. In Chapter 3 we were only concerned with identifying the interactions of pre-existing neighbourhoods. The neighbourhoods were assumed to already be available from some unspecified source (e.g. from previous experience, part of a library, described in the literature, etc.). Rather than relying on the properties of these neighbourhoods, one approach could be to actively create neighbourhoods with desirable interaction signatures for a given problem.

Rather than just considering the creation of a single neighbourhood it may be more instructive to consider creating a set of neighbourhoods with complementary properties (i.e. the ability to collectively interact with all the constraints). This would be appear to be a similar problem to that of *Set Packing* (described in Skiena [2008, Sec. 18.2, pp. 625–627]). Potentially generative techniques like GP could be used "*evolve*" neighbourhoods. The Interaction Detector could evaluate the resulting candidates to uncover their behaviour.

Integrating CP techniques with Local Search neighbourhoods has been explored in Vasquez et al. [2003, 2005]. Prestwich looks at the same issue of using Forward Checking [2002c], maintaining Arc Consistency [2002b], and generally hybridising Local Search with pruning techniques [2002a]. This topic is also covered independently in Dechter [2003, Sec. 7.3, pp. 198–205] where the focus is on using Local Search to make decisions that result in problems which are resolvable via consistency techniques. A *cycle-cutset* is the collection of variables (in a constraint network) that—once assigned—allow the remaining variables' values to be determined solely via arc-consistency. Dechter describes interleaving a Local Search phase (where cutset variables are assigned) with a propagation phase.

5.2.2 Relationships between Neighbourhoods

In the previous section we discussed creating neighbourhoods (and groups of neighbourhoods) with desirable interaction properties. This section proposes focusing on the relationships between neighbourhoods themselves, without reference to interaction properties. By relationships, what we really mean are the *set relationships* (e.g. subset $\subseteq$, strict subset $\subset$) between the collections of neighbours that neighbourhoods generate. If from a given starting assignment, a, two neighbourhoods, $N_1(a)$ and $N_2(a)$, produce the neighbourhood sets $\{a_1, a_2, a_3\}$ and $\{a_1, a_2, a_3, a_4\}$, then we could conclude that $N_1(a) \subset N_2(a)$. These associations are evident in some of our framework's neighbourhoods (e.g. `validRoomAssignments` $\subset$ `roomAssignments`). More complex relationships also exist; in Appendix A we note that `ConsistentAllSwaps<LS>` $=$ `ConsistentSingleDayAllSwaps<LS>` $\cup$ `ConsistentInterDayAllSwaps<LS>`.

As the neighbourhood designer, identifying these relationships is relatively straightforward. However, as we have been promoting treating neighbourhoods as reusable components it would be desirable to be able to detect the relationships between neighbourhoods automatically. Some of the techniques used in the Interaction Detector would be reusable. For example, querying a neighbourhood for its variables could quickly identify unrelated neighbourhoods.

Classification of the *nature* of a neighbourhood's permutations may also prove useful. By *nature* we mean the type of assignment being performed (e.g. is it an exchange, a reassignment, or multiples thereof?). In our current scheme a neighbourhood is a black-box that operates as a function over a collection of variables; we can query its size, but the nature of its permutations is hidden and only revealed to us via their effect on the violation state of the constraint families. In Ågren et al.'s work on CON there are five operators that can be combined (using set operators) to describe a neighbourhood. The nature of a neighbourhood can be captured precisely but it does require a more structured approach to neighbourhood construction than systems like COMET provide.

Whilst uncovering structure within collections of neighbourhoods is pleasing (in a purely academic sense), there could also be important practical considerations. When operating with multiple neighbourhoods, ideally each neighbourhood should create distinct sets of neighbours. It is inefficient to have more than one neighbourhood generating each neighbour. Neighbourhood relationship information could be used to perform neighbourhood replacements. In CP the TAILOR system by Rendl [2010] automatically reformulates existing constraint models to remove common subexpressions (which lead to less efficient models and primarily arise from naïve modelling decisions). Our work could be viewed as the inverse of common subexpression removal; rather than exploring a single large neighbourhood, a search could choose to explore the equivalent conjunction of smaller neighbourhoods.

5.2.3 Improving the Interaction Detector

In Section 11 of Chapter 3 we set out the limitations of the Interaction Detector. The chief of these is that, in most situations, the Interaction Detector is *incomplete*. It would be desirable to mitigate this weakness, especially if we were potentially using it during the neighbourhood creation phase. The empirical results showed that it rarely made misclassifications; however, it would be better if its failure to find an interaction could be taken as definitive. The Interaction Detector uses the same neighbourhoods (and exploration method) as any subsequent Local Search procedure. How to guarantee completeness using the current architecture remains an open question. It is not obvious how one could explore the neighbourhoods and guarantee completeness without sacrificing their black-box nature. It may be that techniques from the SAT / CP fields such as *clause-learning*, *nogoods* or *explanations* could be incorporated.

Another aspect of the Interaction Detector that would be beneficial to improve is its sensitivity to the *direction* of the interactions it uncovers. Any change of violations is classified as an interaction. If a neighbourhood can only reduce violations, then this potentially useful information is lost. Using the current detection strategy it would be possible to determine the direction of the interactions. The unresolved issue would be how to determine if the interaction is only in a single direction. The simplest strategy would be to perform multiple detections for each constraint-neighbourhood pair. If the Detector encountered an interaction (in a direction) that differs from its previous observations then you could safely conclude the neighbourhood was capable of both violating and satisfying that constraint. If all the discovered observations are in a single direction then the default position could be to assume that the relationship is also just in that direction.

At present the Interaction Detector operates as an independent pre-processing step. The model feedback (described in Section 4.4) is also a stand-alone component (which depends on the interaction information from the Interaction Detector). These could both be more tightly connected so that user gets modelling feedback as part of the interaction detection process. We envisage that this could behave in a similar manner to the compiler warnings generated whilst developing software.

5.2.4 Creating Multi-phase Algorithms

In Ågren et al. [2007a, Sec. 5] they explore potential distributions of constraints to phases in the Progressive Party Problem via an empirical series of experiments. They hypothesise that such an automatic phase partitioning may be possible, via static analysis, but offer no advice on how this could be performed. Their work builds upon the earlier work of Minton [1996] who produced the MULTI-TAC system for the configuration of CSP solvers. We hypothesise that the constraint-neighbourhood interaction information could be used for this purpose.

The constraint-neighbourhood interactions form a bipartite graph. A bipartite graph is a graph with the property that all its nodes can be divided in two distinct sets; none of the nodes within one group have any edges connecting them to any other node within the same group. Bipartite graphs can be found in a number of real-world situations, typically recognising some association between two differing classes of objects e.g. linking keys to the locks they open. In the constraint-neighbourhood interaction bipartite graph one set of nodes are neighbourhoods and the other set are constraint families; Fig 3.12 showed a visual representation of these connections. In CSPs the relationship between variables and their domains forms a bipartite graph. It was this bipartite property that allowed the creation of efficient pruning techniques for the `alldifferent` constraint by Régin [1994]. Finding a feasible solution for the `alldifferent` constraint is equivalent to finding a maximum matching in the variable-value bipartite graph. A matching is a set of edges that do not share any common vertices. A *maximum matching* is a matching that contains the largest possible number of edges. Finding maximum matchings in bipartite graphs is a well-studied problem for which there are efficient algorithms.

One way to find a maximum matching is by treating the bipartite graph as a network flow graph. The bipartite graph can be converted to a network flow by creating a *source* node that is connected to all the nodes in the first group. Similarly, all the other nodes are connected to a single *sink* node. These additional nodes allow the matchings to be found by calculating the maximum flow. The flow graph also allows the identification of Strongly Connected Components (SCCs). A directed graph is *strongly connected* if it is possible to find a path from every node to every other node. By identifying the Strongly Connected Components (SCCs) (which can be done using DFS in polynomial time) then replacing each SCC in the graph with a single node—whilst retaining the edges between components—creates a Directed Acyclic Graph (DAG). In the context of the `alldifferent` constraint, the edges which cross between SCCs can be pruned because they represent assignments which reduce the possible choices from a self-contained subsection. Directed Acyclic Graphs (DAGs) occur naturally in scheduling applications where they are used to capture the precedences between jobs. Any valid schedule must have an acyclic precedence graph (otherwise it would indicate that somewhere a task is required to be its own predecessor). Each job becomes a vertex and an edge represents that the source job must be completed before the destination job. Constructing a feasible ordering of a DAG's vertices (that respects all the precedence constraints) can be done by performing a polynomial time *topological sort*. The DAG formed from the constraint-neighbourhood interaction graph represents a partitioning of the search into phases. Each vertex in the DAG is *matching* between a subset of the neighbourhoods and constraints. The edges between vertices in the DAG reflect that some neighbourhoods interact with constraints in other connected components. By topologically sorting the DAG to respect the precedence constraints, the resulting phase ordering means that earlier phases will not introduce violations in the earlier phases.

In a conventional COP situation the user will choose to partition the search such that the hard constraints are optimised before the soft constraints. The

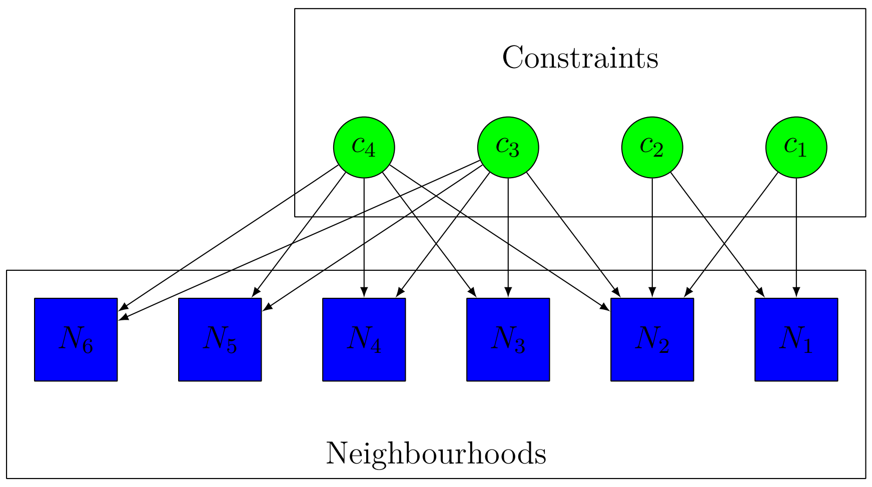


Figure 5.1: A constraint-neighbourhood interactions graph.

soft constraint optimisation phase should retain feasibility. That means that the soft constraint phase cannot contain any neighbourhoods that interact with any of the hard constraints (i.e. the precedence graph must be *acyclic*). The benefit of this strict two-phase strategy is that once the search finds a feasible solution any subsequent improvements will still be feasible. However, this artificial ordering may not be the most efficient way to solve the problem. Our search partitioning creates an ordering that more accurately reflects the relationships between neighbourhoods and tries to structure the search so that it progresses in a smooth fashion. The most disruptive neighbourhoods are used at the start before switching to more focused neighbourhoods to optimise the remaining constraints without undoing the previous search effort.

The GLSM structure at the core of the CDVNS algorithm in Chapter 4 means that any DAG resulting from this process would be directly executable. The transitions between each phase are guarded by the conditions that all the previous constraints are satisfied. This structure also opens up several interesting questions about the behaviour when reaching plateaux or optima. Should the search use the existing strategies of disruption, dissuasion, and heuristic manipulation? Could tree search strategies prove useful? For instance, if the search has plateaued, could back-tracking to a previous phase widen the scope of the search and allow progress? Alternatively, maybe some form of limited *dives* into successor phases (like TS's *strategic oscillation*) would lead the search in a more fruitful direction. This would be almost like an inversion of LNS. In LNS subsections of a problem are chosen by a Local Search and then optimised by a CP tree search; in this

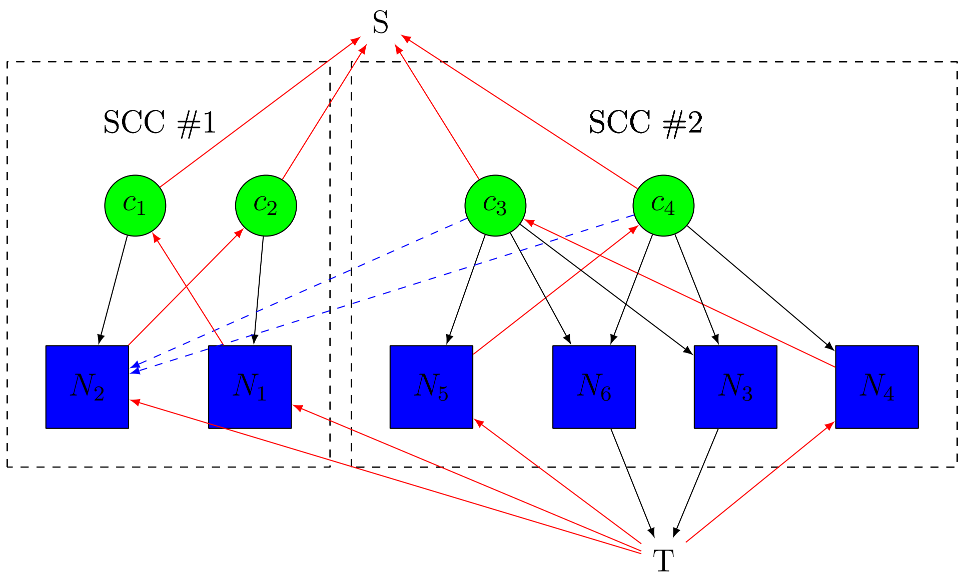


Figure 5.2: The interaction graph from Fig. 5.1 as a flow graph highlighting the SCCs. The residual edges are denoted in red with edges between SCCs marked in blue.

situation a tree search is being used to control multiple VNS / Local Searches each focusing on satisfying a subset of the constraints.

Using the 41 neighbourhoods and their interaction signatures from Chapter 3, no distinct structure emerged from the SCC identification. Once the bipartite graph had been converted to a maximal matching problem only one connected component emerged containing all the constraints. Using a subset of the neighbourhoods results in usable partitions, but raises the problem of identifying the appropriate neighbourhoods. Alternatively, the neighbourhoods can have their interactions artificially suppressed (via the *preserved constraints*), but neither strategy reached a satisfactory level of development to properly evaluate the multi-phase generation.

5.3 Final Remarks

This chapter has revisited the objectives set out in the Introduction and connected those to the work carried out in Chapters 3 and 4. It has also explored some potential directions in which we could envisage this work being expanded.

The central theme has been about exploring the connection between the neighbourhoods used by a Local Search and CSPs' constraints. CBLS's incorporation of

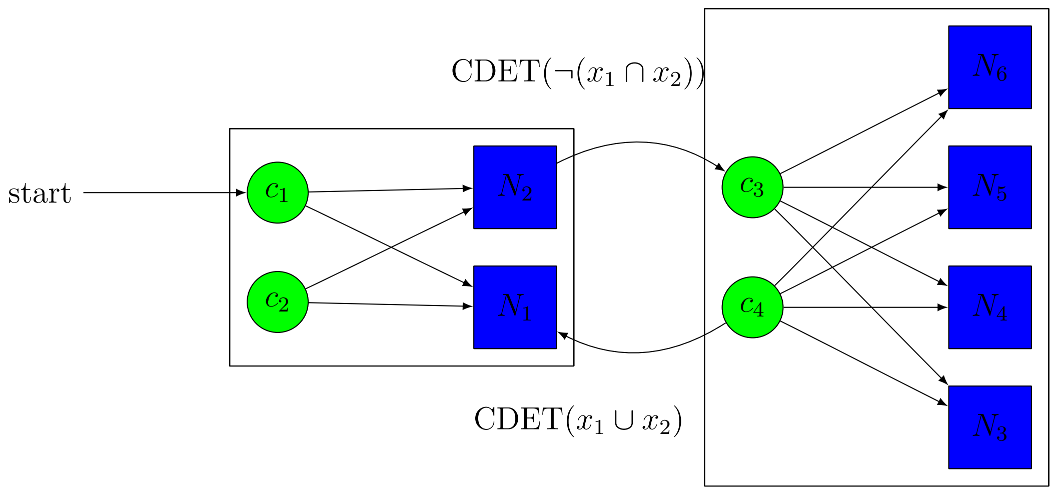


Figure 5.3: The ordering resulting from treating Fig. 5.2's SCCs as nodes in a DAG.

a model structure into Local Search has brought it closer to other search technologies like CP and MIP. Previously, Local Search algorithms were typically tightly coupled to the individual model being solved; CBLS provides an opportunity to relax that coupling and create more reusable systems where the same solver can be applied to multiple problems. However, Local Search's effectiveness on CSPs has been due to its ability to exploit problem structure effectively. We have explored how generic information about the behaviour of neighbourhoods can be extracted and subsequently exploited in that CBLS context.

The extraction aspect of the work (from Chapter 3) created definitions of what we sought to uncover, described a framework that allowed it to operate, proposed a detection method and showed its effectiveness. This was the more successful component of our work. The second theme about how to use the extracted information failed to uncover any strong benefits to the approaches we tried. We looked at using the constraint-neighbourhood information to avoid futile exploration and also guide the overall search direction. Our experiments did not show that either provided any significant advantages.

Ultimately, we hope that this work contributes to the further study of CBLS and will aid its continued formalisation and adoption. Mathematical Programming techniques have been around since the early days of OR. CP has been a practical problem-solving tool since its emergence from the AI field in the 1980s. Both have active communities that ensure they continue to be refined, improved, and most importantly *used*. CBLS is a newer addition to the problem-solver's arsenal, but there is every reason to believe it can and will achieve the same level of maturity and recognition as its counterparts.

Bibliography

1st International Timetabling Competition, October 2002. URL `http://www.idsia.ch/Files/ttcomp2002/`. Cited on page 104.

Emile Hubertus Leonardus Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley and Sons, Baffins Lane, Chichester, West Sussex, PO19 1UD, England, 1997. Cited on pages 199 and 201.

Magnus Ågren. *Set Constraints for Local Search*. PhD thesis, Department of Information Technology, Uppsala University, Sweden, December 2007. Cited on pages 26, 59, 60, 64, 65, 68, and 234.

Magnus Ågren, Pierre Flener, and Justin Pearson. Set Variables and Local Search. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 19–33. Springer Berlin / Heidelberg, May 2005. doi:10.1007/11493853_4. Cited on page 68.

Magnus Ågren, Pierre Flener, and Justin Pearson. On Constraint-Oriented Neighbours for Local Search. Technical Report 2007-009, Department of Information Technology, Uppsala University, Box 337, SE – 751 05 Uppsala, Sweden, March 2007a. Cited on pages 59, 68, 183, and 184.

Magnus Ågren, Pierre Flener, and Justin Pearson. Generic Incremental Algorithms for Local Search. *Special Issue on Local Search Techniques for Constraint Satisfaction, Constraints*, 12(3):293–324, September 2007b. doi:10.1007/s10601-007-9021-0. Cited on page 68.

Magnus Ågren, Pierre Flener, and Justin Pearson. Revisiting constraint-directed search. *Information and Computation*, 207(3):438–457, March 2009. doi:10.1016/j.ic.2008.12.001. Cited on pages 59 and 68.

Ravindra Kumar Ahuja, James Berger Orlin, and Dushyant Sharma. Very large-scale neighborhood search. *International Transactions in Operations Research*, 7(4–5):301–317, September 2000. doi:10.1111/j.1475-3995.2000.tb00201.x. Cited on page 26.

Ravindra Kumar Ahuja, Özlem Ergun, James Berger Orlin, and Abraham P. Punnen. A Survey of Very Large Scale Neighborhood Search Techniques. *Discrete Applied Mathematics*, 123(1–3):75–102, November 2002. doi:10.1016/S0166-218X(01)00338-9. Cited on page 26.

Alexandre Albino Andreatta, Sérgio Eduardo Rodrigues de Carvalho, and Celso Carneiro Ribeiro. A Framework for Local Search Heuristics for Combinatorial Optimization Problems. In Voß and Woodruff [2002], chapter 3, pages 59–80. doi:10.1007/0-306-48126-X_3. Cited on page 61.

Alastair Andrew. Automatically Detecting Neighbourhood Constraint Interactions using Comet. In Kostas Stergiou and Roland Yap, editors, *Proceedings of the CP 2008 Doctoral Programme*, pages 7–12, September 2008. Cited on page 10.

Alastair Andrew. Exploiting Constraint-Neighbourhood Interactions. In Frank Hutter and Marco A. Montes de Oca, editors, *Proceedings of the Doctoral Symposium on Engineering Stochastic Local Search Algorithms (SLS-DS 09)*, TR/IRIDIA/2009-024, pages 41–45, Université Libre De Bruxelles, Av F. D. Roosevelt 50, CP 194/6, September 2009. Cited on pages .

Alastair Andrew and John Levine. Automatically Detecting Neighbourhood Constraint Interactions using Comet. In Yehuda Naveh and Pierre Flener, editors, *Proceedings of the 5th International Workshop on Local Search Techniques in Constraint Satisfaction*, Sydney, Australia, September 2008. Cited on page 10.

Alastair Andrew, John Levine, and Derek Long. Constraint Directed Variable Neighbourhood Search. In Yehuda Naveh and Andrea Roli, editors, *Proceedings of the 4th International Workshop on Local Search Techniques in Constraint Satisfaction*, Providence, Rhode Island, USA, September 2007. Cited on page 10.

Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1st edition, 2007. Cited on page 62.

Roman Barták. On-line Guide to Constraint Programming, 1998. URL `http://ktiml.mff.cuni.cz/~bartak/constraints/`. Cited on page 53.

Roberto Battiti and Giampietro Tecchiolli. The Reactive Tabu Search. *Operations Research Society of America (ORSA) Journal on Computing*, 6(2):126–140, Spring 1994. doi:10.1287/ijoc.6.2.126. Cited on page 45.

Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations Research / Computer Science Interfaces Series*. Springer US, 1st edition, November 2008. doi:10.1007/978-0-387-09624-7. Cited on page 45.

John E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operations Research Society*, 41(11):1069–1072, November 1990. doi:10.1057/jors.1990.166. URL `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`. Cited on page 50.

Stefano Benedettini, Andrea Roli, and Luca Di Gaspero. EasyGenetic: A Template Metaprogramming Framework for Genetic Master-Slave Algorithms. In Thomas Stützle, Mauro Birattari, and Holger Hendrik Hoos, editors, *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics. International Workshop, SLS 2009, Brussels, Belgium, September 3–4, 2009*, volume 5752 of *Lecture Notes in Computer Science*, pages 135–139. Springer Berlin / Heidelberg, September 2009. doi:10.1007/978-3-642-03751-1_-14. Cited on page 62.

Norman Linstead Biggs, E. Keith Lloyd, and Robin James Wilson. *Graph Theory, 1736–1936*. Clarendon Press, Oxford University Press, Ely House, London, W. 1, 1st edition, 1976. Cited on page 12.

Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günther Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund Keiran

Burke, and Natasa Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 11–18, New York, July 2002. Morgan Kaufmann Publishers. Cited on page 45.

Markus Bohlin. *Design and Implementation of a Graph-Based Constraint Model for Local Search.* PhD thesis, Department of Computer Science and Engineering, Mälardalen University, Västerås, Sweden, April 2004. Cited on pages 61 and 68.

Sally C. Brailsford, Peter M. Hubbard, Barbara Mary Smith, and H. Paul Williams. Organizing a social event–a difficult problem of combinatorial optimization. *Computers & Operations Research*, 23(9):845–856, September 1996. doi:10.1016/0305-0548(96)00001-9. Cited on page 68.

Edmund Keiran Burke, Yuri Bykov, James Newall, and Sanja Petrovic. A Time-Predefined Local Search Approach to Exam Timetabling Problems. Technical Report NOTTCS-TR-2001-6, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB, UK, 2001. Cited on page 107.

Edmund Keiran Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In Glover and Kochenberger [2003], chapter 16, pages 457–474. doi:10.1007/0-306-48056-5_16. Cited on page 45.

Edmund Keiran Burke, Adam Eckersley, Barry McCollum, Sanja Petrovic, and Rong Qu. Hybrid Variable Neighbourhood Approaches to University Exam Timetabling. *European Journal of Operational Research*, 206(1):46–53, October 2010. doi:10.1016/j.ejor.2010.01.044. Cited on pages 100 and 137.

Yuri Bykov. The Description of the Algorithm for International Timetabling Competition. Technical report, University of Nottingham, March 2003. Cited on page 107.

Sebastien Cahon, Nouredine Melab, and El-Ghazali Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, May 2004. doi:10.1023/B:HEUR.0000026900.92269.ec. Cited on page 62.

Marco Chiarandini, Krzysztof Socha, Mauro Birattari, and Olivia Rossi-Doria. International Timetabling Competition: A Hybrid Approach. Technical report, March 2003. Cited on pages 107, 114, and 222.

Marco Chiarandini, Mauro Birattari, Krzysztof Socha, and Olivia Rossi-Doria. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling*, 9(5):403–432, October 2006. doi:10.1007/s10951-006-8495-8. Cited on pages 107 and 222.

Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In Kathleen Steinhöfel, editor, *Stochastic Algorithms: Foundations and Applications*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90. Springer Berlin / Heidelberg, December 2001. doi:10.1007/3-540-45322-9_5. Cited on page 54.

Richard K. Congram, Chris N. Potts, and Steef L. van de Velde. An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem. *INFORMS Journal on Computing*, 14(1):52–67, Winter 2002. doi:10.1287/ijoc.14.1.52.7712. Cited on page 27.

Stephen Arthur Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, United States*, pages 151–158, New York, 1971. Association for Computing Machinery. doi:10.1145/800157.805047. Cited on page 39.

Jean-François Cordeau, Brigitte Jaumard, and Rodrigo Morales. Efficient Timetabling Solution with Tabu Search. Technical report, March 2003. Cited on page 107.

G. A. Croes. A Method For Solving Traveling-Salesman Problems. *Operations Research*, 6(6):791–812, November–December 1958. doi:10.1287/opre.6.6.791. Cited on pages 23 and 24.

Tim Curtois, Laurens Fijn van Draat, Jan-Kees van Ommeren, and Gerhard Post. Progress Control in Variable Neighbourhood Search. In Edmund Keiran Burke and Hana Rudová, editors, *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2006)*, pages 376–380, Brno, Czech Republic, 2006. Faculty of Informatics, Masaryk University. Cited on pages 135 and 149.

Martin Davis, George Logemann, and Donald William Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, July 1962. doi:10.1145/368273.368557. Cited on page 235.

Michael de la Maza and Deniz Yuret. Dynamic Hill Climbing. *AI Expert*, 9(3), 1994. Cited on page 36.

Rina Dechter. *Constraint Processing*. An Imprint of Elsevier Science. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2003. Cited on pages 53, 66, and 182.

Luca Di Gaspero and Andrea Schaerf. Multi-Neighbourhood Local Search with Application to Course Timetabling. In Edmund Keiran Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling IV*, number 2740 in Lecture Notes in Computer Science, pages 263–278. Springer-Verlag, Berlin-Heidelberg, Germany, September 2003a. doi:10.1007/b11828. Cited on pages 26 and 107.

Luca Di Gaspero and Andrea Schaerf. Timetabling Competition TTComp 2002: Solver Description. Technical report, Università di Udine, March 2003b. Cited on page 107.

Luca Di Gaspero and Andrea Schaerf. EasyLocal++: an object-oriented framework for flexible design of local search algorithms. *Software – Practice & Experience*, 33(8):733–765, July 2003c. doi:10.1002/spe.524. Cited on page 61.

Luca Di Gaspero and Andrea Schaerf. Neighborhood Portfolio Approach for Local Search applied to Timetabling Problems. *Journal of Mathematical Modeling and Algorithms*, 5(1):65–89, April 2006. doi:10.1007/s10852-005-9032-z. Cited on page 107.

Luca Di Gaspero and Andrea Schaerf. EasySyn++: A Tool for Automatic Synthesis of Stochastic Local Search Algorithms. In Thomas Stützle, Mauro Birattari, and Holger Hendrik Hoos, editors, *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics. SLS 2007*, volume 4638 of *Lecture Notes in Computer Science*, pages 177–181, Berlin, Germany, 2007. Springer Berlin / Heidelberg. doi:10.1007/978-3-540-74446-7_13. Cited on page 62.

Edsger Wybe Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi:10.1007/BF01386390. Cited on pages 19 and 21.

Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In Yves Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence, ECAI 88, Munich, Germany, August 1–5, 1988*, pages 290–295. Pitmann Publishing, August 1988. Cited on page 106.

Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, Cambridge, Massachusetts 02142, July 2004. Cited on page 43.

Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Italy, June 1991. Cited on page 43.

Raphaël Dorne and Christos Voudouris. HSF: the iOpt's framework to easily design Meta-Heuristic methods. In Mauricio G. C. Resende, Jorge Pinho de Sousa, and Ana Viana, editors, *Metaheuristics: Computer Decision-Making*, volume 86 of *Applied Optimization*, chapter 11, pages 237–256. Kluwer Academic Publishers, Norwell, MA, USA, 2004. Cited on page 61.

György Dósa. The Tight Bound of First Fit Decreasing Bin-Packing is $FFD(I) \leq \frac{11}{9}OPT(I) + \frac{6}{9}$. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *Lecture Notes in Computer Science*, chapter 1, pages 1–11. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-74450-4_1. Cited on page 23.

Gunter Dueck. New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel. *Journal of Computational Physics*, 104(1):86–92, January 1993. doi:10.1006/jcph.1993.1010. Cited on pages 34 and 236.

Gunter Dueck and Tobias Scheuer. Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. *Journal of Computational Physics*, 90(1):161–175, September 1990. doi:10.1016/0021-9991(90)90201-B. Cited on page 34.

Pham Quang Dung, Yves Deville, and Pascal Van Hentenryck. LS(Graph & Tree): A Local Search Framework for Constraint Optimization on Graphs and Trees. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC 09)*, volume 2, pages 1402–1407. ACM, March 2009. doi:10.1145/1529282.1529595. Cited on page 83.

Comet Tutorial. Dynamic Decision Technologies Inc., One Richmond Square, Providence, RI 02906, United States, 2.0 edition, March 2010. Cited on pages 56 and 106.

Marc Ebner, Mark Shackleton, and Rob Shipman. How Neutral Networks Influence Evolvability. *Complexity*, 7(2):19–33, November/December 2001. doi:10.1002/cplx.10021. Cited on page 80.

Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Imperialis Scientiarum Petropolitanae*, 8:128–140, 1736. Cited on pages 12 and 14.

Hai Fang and Wheelr Ruml. Complete Local Search for Propositional Satisfiability. In Deborah Louise McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25–29, 2004, San Jose, California, USA*, pages 161–166. AAAI Press / The MIT Press, 2004. ISBN 0-262-51183-5. Cited on page 32.

Thomas A. Feo and Mauricio G. C. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8(2):67–71, April 1989. doi:10.1016/0167-6377(89)90002-3. Cited on pages 37 and 236.

Andreas Fink and Stefan Voß. HotFrame: A heuristic optimization framework. In Voß and Woodruff [2002], chapter 4, pages 81–154. doi:10.1007/0-306-48126-X_4. Cited on page 61.

Andreas Fink, Stefan Voß, and David L. Woodruff. Metaheuristic Class Libraries. In Glover and Kochenberger [2003], chapter 18, pages 515–535. doi:10.1007/0-306-48056-5_18. Cited on page 62.

Merrill Meeks Flood. The Traveling-Salesman Problem. *Operations Research*, 4 (1):61–75, February 1956. doi:10.1287/opre.4.1.61. Cited on page 23.

Lawrence J. Fogel, Alvin Jewel Owens, and Michael John Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley and Sons Ltd., New York, USA, 1966. Cited on page 41.

Mark S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 1983. Cited on pages 54 and 59.

Mark S. Fox. Constraint-Guided Scheduling—A Short History of Research at CMU. *Computers in Industry*, 14(1–3):79–88, May 1990. doi:10.1016/0166-3615(90)90107-Z. Cited on page 54.

Michael Randolph Garey and David Stifler Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, New York, 1979. Cited on page 84.

Alfonso Emilio Gerevini, Alessandro Saetti, and Mauro Vallati. An Automatically Configurable Portfolio-Based Planner with Macro-Actions: PbP. In Alfonso Emilio Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, May 19–23, 2009*, pages 350–353, Menlo Park, California, May 2009. AAAI Press. Cited on page 46.

Fred Glover. Tabu Search—Part I. *Operations Research Society of America (ORSA) Journal on Computing*, 1(3):190–206, Summer 1989. doi:10.1287/ijoc.1.3.190. Cited on pages 35 and 36.

Fred Glover. Tabu Search—Part II. *Operations Research Society of America (ORSA) Journal on Computing*, 2(1):4–32, Winter 1990. doi:10.1287/ijoc.2.1.4. Cited on page 35.

Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, Boston, MA, USA, 1997. Cited on pages 35, 55, 98, and 165.

Fred W. Glover. Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems. *Discrete Applied Mathematics*, 65 (1–3):223–253, March 1996. doi:10.1016/0166-218X(94)00037-E. Cited on pages 26 and 221.

Fred W. Glover and Gary A. Kochenberger, editors. *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science.* Kluwer Academic Publishers, 2003. doi:10.1007/b101874. Cited on pages 193, 197, 199, 202, and 209.

Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-Tailed Distributions in Combinatorial Search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin / Heidelberg, October 1997. doi:10.1007/BFb0017434. Cited on page 39.

Pierre Hansen and Nenad Mladenović. Variable Neighborhood Search. In Glover and Kochenberger [2003], chapter 6, pages 145–184. doi:10.1007/0-306-48056-5_-6. Cited on page 47.

Pierre Hansen and Nenad Mladenović. First vs. Best Improvement: An Empirical Study. *Discrete Applied Mathematics*, 154(5):802–817, April 2006. doi:10.1016/j.dam.2005.05.020. Cited on page 28.

Robert Harder. OpenTS: An Open Source Java Tabu Search Framework. Presented at the INFORMS Annual Meeting, Miami, November 2001. URL `http://www.coin-or.org/Ots/`. Cited on page 61.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. doi:10.1109/TSSC.1968.300136. Cited on page 21.

Barbara Hayes-Roth and Richard Earl Korf, editors. *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, Washington, USA, July–August 1994. AAAI Press. Cited on pages 206 and 207.

John Leroy Hennessy and David Andrew Patterson. *Computer Architecture: A Quantitative Approach.* An Imprint of Elsevier. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 3rd edition, 2003. Cited on page 27.

Alain Hertz, Éric D. Taillard, and Dominique de Werra. Tabu Search. In Aarts and Lenstra [1997], chapter 5, pages 121–136. Cited on page 36.

Geoffrey Everest Hinton. Learning Distributed Representations of Concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Societ.y*, pages 1–12, Amherst, Mass., 1986. Cited on page 109.

Brahim Hnich, Barbara Mary Smith, and Toby Walsh. Dual Modelling of Permutation and Injection Problems. *Journal of Artificial Intelligence Research*, 21: 357–391, January–June 2004. doi:10.1613/jair.1351. Cited on page 93.

John Henry Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975. Cited on page 41.

Robert Craig Holte and Adele E. Howe, editors. *Proceedings of the 22nd AAAI Conference on Artificial Intelligence, July 22–26, 2007, Vancouver, British Columbia, Canada*, Menlo Park, California, July 2007. AAAI Press. Cited on pages 200 and 208.

Holger Hendrik Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 1st edition, 2005. Cited on pages 24, 30, 32, 63, 133, 134, and 236.

Bin Hu and Günther R. Raidl. Variable Neighborhood Descent with Self-Adaptive Neighborhood-Ordering. In Carlos Cotta, Antonio J. Fernández, and Jose E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive and Multi-Level Metaheuristics*, Malaga, Spain, 2006. Cited on page 149.

Frank Hutter, Youssef Hamadi, Holger Hendrik Hoos, and Kevin Leyton-Brown. Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms. In Frédéric Benhamou, editor, *Proceedings of the 12th International Conference on the Principles and Practice of Constraint Programming - CP 2006, Nantes, France, September 25–29, 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin / Heidelberg, September 2006. doi:10.1007/11889205_17. Cited on page 45.

Frank Hutter, Holger Hendrik Hoos, and Thomas Stützle. Automatic Algorithm Configuration based on Local Search. In Holte and Howe [2007], pages 1152–1157. Cited on page 45.

Frank Hutter, Holger Hendrik Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009. doi:10.1613/jair.2861. Cited on pages 45 and 99.

Toshihide Ibaraki, Koji Nonobe, and Mutsunori Yagiura, editors. *Metaheuristics: Progress as Real Problem Solvers*, volume 32 of *Operations Research / Computer Science Interfaces Series*. Springer, 2005. doi:10.1007/b107306. Cited on page 209.

Anant Singh Jain, Balasubramanian Rangaswamy, and Sheik Meeran. New and "Stronger" Job-Shop Neighbourhoods: A Focus on the Method of Nowicki and Smutnicki (1996). *Journal of Heuristics*, 6(4):457–480, September 2000. doi:10.1023/A:1009617209268. Cited on page 48.

Chris Johnson and Keith Johnson. TimeTabler, February 1998. URL `http://www.timetabler.com/`. Cited on page 102.

David Stifler Johnson and Lyle Andrew McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. In Aarts and Lenstra [1997], chapter 8, pages 215–310. Cited on page 16.

David E. Joslin and David P. Clements. "Squeaky Wheel" Optimization. *Journal of Artificial Intelligence Research*, 10:353–373, January–June 1999. doi:10.1613/jair.561. Cited on pages 38 and 55.

Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002. doi:10.1016/S0004-3702(02)00221-7. Cited on page 54.

Scott Kirkpatrick, Charles Daniel Gelatt Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983. doi:10.1126/science.220.4598.671. Cited on page 34.

Joshua D. Knowles and Richard A. Watson. On the Utility of Redundant Encodings in Mutation-Based Evolutionary Search. In Juan Juliá Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, José-Luis Fernández-Villacañas, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 88–98. Springer Berlin / Heidelberg, September 2002. doi:10.1007/3-540-45712-7_9. Cited on page 80.

Philipp A. Kostuch. The University Course Timetabling Problem with a Three-Phase Approach. In Edmund Keiran Burke and Michael Alan Trick, editors, *Practice and Theory of Automated Timetabling V*, volume 3616 of *Lecture Notes* *in Computer Science*, pages 109–125, Heidelberg, November 2004. Spring Berlin. doi:10.1007/11593577_7. Cited on page 107.

François Laburthe and Yves Caseau. SALSA: A Language for Search Algorithms. *Constraints*, 7(3–4):255–288, July 2002. doi:10.1023/A:1020565317875. Cited on page 63.

Hoong Chuin Lau, Wee Chong Wan, Steven Halim, and Kaiyang Toh. A software framework for fast prototyping of meta-heuristics hybridization. *International Transactions in Operations Research*, 14(2):123–141, March 2007. doi:10.1111/j.1475-3995.2007.00578.x. Cited on page 62.

Vianney le Clément, Yves Deville, and Christine Solnon. Constraint-Based Graph Matching. In Ian Philip Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 274–288. Springer Berlin / Heidelberg, September 2009. doi:10.1007/978-3-642-04244-7_23. Cited on page 83.

Rhydian Marc Rhys Lewis. *Metaheuristics for University Course Timetabling*. PhD thesis, Centre for Emergent Computing, School of Computing, Edinburgh Napier University, August 2006. Cited on page 107.

Shen Lin. Computer Solutions of the Traveling-Salesman Problem. *Bell System Technical Journal*, 44:2245–2269, July–December 1965. Cited on pages 24 and 38.

Shen Lin and Brian W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, March–April 1973. Cited on page 24.

Samir Loudni, Patrice Boizumault, and Nicolas Levasseur. Advanced generic neighborhood heuristics for VNS. *Engineering Applications for Artificial Intelligence*, 23(5):736–764, August 2010. doi:10.1016/j.engappai.2010.01.014. Cited on page 165.

Helena Ramalhinho Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated Local Search. In Glover and Kochenberger [2003], chapter 11, pages 321–353. doi:10.1007/0-306-48056-5_11. Cited on page 32.

Zhipeng Lü, Jin-Kao Hao, and Fred Glover. Neighborhood Analysis: A Case Study on Curriculum-Based Course Timetabling. *Journal of Heuristics*, 2010. doi:10.1007/s10732-010-9128-0. (In Press). Cited on page 108.

Mirko Maischberger. METSlib metaheuristic framework version 0.4.2, July 2009. URL `https://projects.coin-or.org/metslib`. Cited on page 62.

Thierry Mautor. Intensification Neighborhoods for Local Search Methods. In Ribeiro and Hansen [2002], chapter 22, pages 493–508. Cited on page 27.

Thierry Mautor and Philippe Michelon. Mimausa: a new hybrid method combining exact solution and local search. In *Proceedings of the 2nd International Conference on Metaheuristics (MIC-97)*, pages 15–16, Sophia-Antipolis, France, 1997. Cited on pages 27 and 55.

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21(6):1087–1092, June 1953. doi:10.1063/1.1699114. Cited on page 34.

Laurent Michel. *Localizer: A Modeling Language for Local Search.* PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island, USA, October 1998. Cited on page 56.

Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1–2):43–84, January 2000. doi:10.1023/A:1009818401322. Cited on pages 56 and 63.

Laurent Michel and Pascal Van Hentenryck. Localizer++: An Open Library for Local Search. Technical Report CS-01-02, Department of Computer Science, Brown University, Providence, Rhode Island 02912, January 2001. Cited on page 56.

Laurent Michel and Pascal Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 83–100, Seattle, Washington, USA, November 2002. Cited on page 68.

Steven Minton. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1(1–2):7–43, September 1996. doi:10.1007/BF00143877. Cited on pages 45 and 184.

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*, 58(1–3):161–205, December 1992. doi:10.1016/0004-3702(92)90007-K. Cited on pages 54, 65, and 165.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. Bradford Books. The MIT Press, Cambridge, Massachusetts, 1st edition, February 1998. Cited on pages 41 and 42.

Melanie Mitchell, John Henry Holland, and Stephanie Forrest. When Will a Genetic Algorithm Outperform Hill Climbing? In Jack David Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems 6, 7th NIPS Conference, Denver, Colorado, USA*, pages 51–58, San Mateo, CA, 1993. Morgan Kaufmann. Cited on page 42.

Nenad Mladenović and Pierre Hansen. Variable Neighborhood Search. *Computers & Operations Research*, 24(11):1097–1100, November 1997. doi:10.1016/S0305-0548(97)00031-2. Cited on pages 46, 133, and 239.

Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon: Synthesizing Scheduling Algorithms from High-Level Models. In *Proceedings of the 2009 INFORMS Computing Society Conference (ICS09)*, Charleston, South Carolina, January 11–13 2009. Cited on page 83.

Edward Forrest Moore. The Shortest Path Through a Maze. In *Proceedings of an International Symposium on the Theory of Switching*, pages 285–292, Cambridge, Massachusetts, 1959. Harvard University Press. Cited on page 18.

Paul Morris. The Breakout Method For Escaping From Local Minima. In Richard Fikes and Wendy Lehnert, editors, *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 40–45, Washington, DC, USA, July 11–15 1993. The AAAI Press / The MIT Press. Cited on page 54.

Tomáš Müller. UniTime.org, February 2009. URL `http://www.unitime.org/`. Cited on page 102.

Alexander Nareyek. Using Global Constraints for Local Search. In Eugene Charles Freuder and Richard John Wallace, editors, *Constraint Programming and Large*

Scale Discrete Optimization, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, chapter 1, pages 9–28. American Mathematical Society, 2001. Cited on page 57.

Eugeniusz Nowicki and Czeslaw Smutnicki. A Fast Taboo Search Algorithm for the Job Shop Problem. *Management Science*, 42(6):797–813, June 1996. doi:10.1287/mnsc.42.6.797. Cited on page 48.

Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, August 2008. Cited on page 46.

OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`. Cited on page 62.

Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, 31 East 2nd Street, Mineola, N.Y. 11501, 2nd edition, 1998. Cited on page 181.

Gilles Pesant and Michel Gendreau. A View of Local Search in Constraint Programming. In Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, editors, *Principles and Practice of Constraint Programming - CP96*, volume 1118 of *Lecture Notes in Computer Science*, pages 353–366. Springer Berlin / Heidelberg, August 1996. doi:10.1007/3-540-61551-2_86. Cited on page 54.

Vinhthuy Phan, Pavel Sumazin, and Steven Skiena. A Time-Sensitive System for Black-Box Combinatorial Optimization. In David Mark Mount and Clifford Stein, editors, *Algorithm Engineering and Experiments, 4th International Workshop, ALENEX 2002, San Francisco, CA, USA, January 4–5, 2002 Revised Papers*, volume 2409 of *Lecture Notes in Computer Science*, pages 16–28. Springer Berlin / Heidelberg, January 2002. doi:10.1007/3-540-45643-0_2. Cited on page 46.

Steven Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research*, 115(1–4): 51–72, September 2002a. doi:10.1023/A:1021140902684. Cited on page 182.

Steven Prestwich. Maintaining Arc-Consistency in Stochastic Local Search. In *Workshop on Techniques for Implementing Constraint Programming Systems*, 2002b. Cited on page 182.

Steven Prestwich. Coloration neighbourhood search with forward checking. *Annals of Mathematics and Artificial Intelligence*, 34(4):327–340, April 2002c. doi:10.1023/A:1014496509129. Cited on pages 54 and 182.

Steven Prestwich and Andrea Roli. Symmetry Breaking and Local Search Spaces. In Roman Barták and Michela Milano, editors, *Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 273–287. Springer Berlin / Heidelberg, 2005. doi:10.1007/11493853_-21. Cited on pages 55 and 73.

Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In Hayes-Roth and Korf [1994], pages 362–367. Cited on page 185.

Andrea Rendl. *Effective Compilation of Constraint Models*. PhD thesis, School of Computer Science, University of St Andrews, January 2010. Cited on page 183.

Celso Carneiro Ribeiro and Pierre Hansen, editors. *Essays and Surveys in Metaheuristics*, volume 15 of *Operations Research / Computer Science Interfaces Series*. Kluwer Academic Publishers, Boston, 2002. Cited on pages 203 and 208.

Mark Roberts, L. Darrell Whitley, Adele E. Howe, and Laura Barbulescu. Random Walks and Neighborhood Bias in Oversubscribed Scheduling. In Graham Kendall, Lei Lei, and Michael Pinedo, editors, *Proceedings of the 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA-05)*, volume 1, pages 98–106, New York, USA, July 2005. Cited on page 80.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series In Artificial Intelligence. Pearson Education, Inc., Upper Saddle River, New Jersey 07458, 2nd edition, 2003. Cited on pages 18 and 20.

Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving Steel Mill Slab Problems with constraint-based techniques: CP, LNS, and CBLS. *Constraints*, 16(2):125–147, April 2011. doi:10.1007/s10601-010-9100-5. Cited on pages 57 and 58.

Alexander Schrijver. On the history of combinatorial optimization (till 1960). In Karen Aardal, George Lann Nemhauser, and Robert Weismantel, editors, *Handbook of Discrete Optimization*, pages 1–68. Elsevier, Amsterdam, July 2005. Cited on page 16.

Christian Schulte and Peter James Stuckey. Speeding Up Constraint Propagation. In Mark Wallace, editor, *Proceedings of the 10th Intenational Conference on the Principles and Practice of Constraint Programming - CP 2004, Toronto, Canada, September 27–October 1, 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633. Springer Berlin / Heidelberg, September / October 2004. doi:10.1007/b100482. Cited on page 73.

Bart Selman and Henry A. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In Ružena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambéry, France, August 28–September 3, 1993*, volume 1, pages 290–295. Morgan Kaufmann, August / September 1993. Cited on pages 39, 40, and 54.

Bart Selman, Hector Joseph Levesque, and David G. Mitchell. A New Method for Solving Hard Satisfiability Problems. In William Roy Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12–16*, pages 440–446. The AAAI Press / The MIT Press, July 1992. Cited on page 40.

Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In Hayes-Roth and Korf [1994], pages 337–343. Cited on page 40.

Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practise of Constraint Programming - CP 98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin / Heidelberg, October 1998. doi:10.1007/3-540-49481-2. Cited on pages 55 and 237.

Paul Shaw, Bruno De Backer, and Vincent Furnon. Improved Local Search for CP Toolkits. *Annals of Operations Research*, 115(1–4):31–50, September 2002. doi:10.1023/A:1021188818613. Cited on page 62.

Steven Sol Skiena. *The Algorithm Design Manual*. Springer-Verlag, 2nd edition, 2008. doi:10.1007/978-1-84800-070-4. Cited on pages 21 and 181.

Barbara Mary Smith, Sally C. Brailsford, Peter M. Hubbard, and H. Paul Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. *Constraints*, 1(1–2):119–138, September 1996. doi:10.1007/BF00143880. Cited on page 68.

Catherine Soanes, editor. *Paperback Oxford English Dictionary.* Oxford University Press, Great Clarendon Street, Oxford OX2 6DP, 2002. Cited on page 19.

Andrew M. Sutton. An analysis of search landscape neutrality in scheduling problems. In Jörg Hoffmann and Jean-Paul Watson, editors, *Proceedings of the ICAPS 2007 Doctoral Consortium*, September 2007. Cited on page 79.

Éric D. Taillard and Stefan Voß. POPMUSIC: Partial Optimization Metaheuristic Under Special Intensification Conditions. In Ribeiro and Hansen [2002], chapter 27, pages 613–629. Cited on page 55.

Ole Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47, February 2011. URL `http://www.gnu.org/s/parallel`. Cited on page 137.

Edward P. K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London and San Diego, 1993. Cited on page 54.

Edward Rolf Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 2nd edition, 2001. Cited on page 109.

Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Massachusetts, 1st edition, January 1999. Cited on pages 56 and 237.

Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search.* The MIT Press, Cambridge, Massachusetts, 1st edition, August 2005. Cited on pages 63, 68, 75, 76, 84, 87, 96, and 171.

Pascal Van Hentenryck and Laurent Michel. Synthesis of Constraint-Based Local Search Algorithms from High-Level Models. In Holte and Howe [2007], pages 273–279. Cited on pages 73, 94, and 177.

Michel Vasquez, Djamal Habet, and Audrey Dupont. Neighborhood Design by Consistency Checking. In *Actes du cinquième congrès de la société Française de*

Recherche Opérationnelle et d'Aide à la décision (ROADEF 2003), 2003. Cited on page 182.

Michel Vasquez, Audrey Dupont, and Djamal Habet. Consistent Neighbourhood in a Tabu Search. In Ibaraki et al. [2005], chapter 17, pages 369–388. doi:10.1007/0-387-25383-1_17. Cited on pages 54 and 182.

Ana Viana, Jorge Pinho de Sousa, and Manuel A. Matos. Constraint Oriented Neighbourhoods - A New Search Strategy in Metaheuristics. In Ibaraki et al. [2005], chapter 18, pages 389–414. doi:10.1007/0-387-25383-1_18. Cited on pages 58, 59, 64, 65, and 235.

Ana Viana, Jorge Pinho de Sousa, and Manuel A. Matos. Fast solutions for UC problems by a new metaheuristic approach. *Electric Power Systems Research*, 78(8):1385–1395, August 2008. doi:10.1016/j.epsr.2008.01.002. Cited on page 59.

Stefan Voß and David L. Woodruff, editors. *Optimization Software Class Libraries*, volume 18 of *Operations Research / Computer Science Interfaces Series*. Kluwer, Boston, 2002. doi:10.1007/b101931. Cited on pages 191 and 197.

Christos Voudouris. *Guided Local Search for Combinatorial Optimisation Problems.* PhD thesis, Department of Computer Science, University of Essex, Colchester, UK, July 1997. Cited on page 37.

Christos Voudouris and Edward P. K. Tsang. Guided Local Search. Technical Report CSM-247, Department of Computer Science, University of Essex, Colchester, CO4 3SQ, UK, August 1995. Cited on pages 37 and 235.

Christos Voudouris and Edward P. K. Tsang. Guided Local Search. In Glover and Kochenberger [2003], chapter 7, pages 185–218. doi:10.1007/0-306-48056-5_7. Cited on page 37.

Stefan Wagner and Michael Affenzeller. HeuristicLab: A Generic and Extensible Optimization Environment. In *Adaptive and Natural Computing Algorithms, Proceedings of the International Conference in Coimbra, Portugal, 2005*, volume VI, pages 538–541. Springer Vienna, December 2005. doi:10.1007/3-211-27389-1_130. Cited on page 62.

Richard John Wallace and Eugene Charles Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *Proceedings of the 9th Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992. Cited on page 73.

Jean-Paul Watson. The templatized metaheuristics framework. In *Proceedings of the 7th Metaheuristics International Conference (MIC 2007)*, 2007. Cited on page 62.

Lin Xu, Frank Hutter, Holger Hendrik Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, May–August 2008. doi:10.1613/jair.2490. Cited on page 46.

Appendix A

Neighbourhoods Summary

"Acceptable at a dance and invaluable at a shipwreck"

First Headmaster of Stowe School, J. F. Roxburgh's response to a parent's enquiry about the type of pupils he aimed to produce.

Neighbourhoods play a central role within any Local Search algorithm and consequently feature heavily in this thesis. This appendix describes the different types of neighbourhoods that are part of our framework. The first section of this appendix describes the seven abstract neighbourhood classes that form the basis of any neighbourhood implemented within our framework. The abstract classes act as wrappers around COMET's differentiation methods and provide much of the binding between a user's neighbourhood and the logistics of differentiating a move against possibly multiple constraint containers. An abstract neighbourhood acts as a template defining the type of move a neighbourhood could have; it is the concrete neighbourhoods that actually specify the behaviour of a neighbourhood. The second section of this chapter gives a synopsis of the concrete neighbourhoods. The concrete neighbourhoods extend from the abstract neighbourhood classes and can actually be used in applications. The final section of this chapter covers the Candidate Lists used to restrict some neighbourhoods and provide different behaviours from the same basic classes.

A.1 Abstract Neighbourhoods

One of COMET's main strengths is the efficient differentiation methods which it provides for constraints and functions. To evaluate the effect of a potential move you must call the appropriate method from the `Constraint<LS>` interface; these functions are covered in Listing 3.6 from Chapter 3. The differentiation methods can calculate the effect of any neighbourhood move. Currently, there are two problems when trying to use the system for multiple constraint families. Firstly, moves can only be differentiated against a single constraint container, and secondly the user must have collated the data needed to differentiate the move manually.

One of the key aspects of our framework is its ability to handle multiple constraint systems (and other constraint and objective containers). Section 3.3 detailed the framework's dynamic scoping of constraints and the automatic **lookahead** handling. The abstract classes manage both of these features so user neighbourhoods need no alterations to work with multiple constraint containers; the abstract classes also take care of the preserved constraints.

A.1.1 Assignment<LS>

`Assignment<LS>` is the simplest of the abstract neighbourhoods. It provides the template for any neighbourhood that is based upon a single variable reassignment, (i.e. one which would be differentiated with `getAssignDelta(`**`var{int}`**` x , `**`int`**` v)`). As well as `ConstraintSystem<LS>`s it supports `Function<LS>`s and `DisequationSystem<LS>`s. Figure A.1 provides a diagram illustrating this type of move. Any neighbourhood that extends `Assignment<LS>` will be an *atomic neighbourhood*; it would be impossible to simulate the effect of an `Assignment<LS>` with a simpler neighbourhood. The maximum size of an `Assignment<LS>` neighbourhood operating over an array of n decision variables would be $\left(\sum_{i=0}^{n-1} Domain_i\right) - 1$. The final subtraction of one is because we make the assumption that a neighbourhood will never offer the current state as a potential neighbour.

A.1.2 MultipleAssignment<LS>

`MultipleAssignment<LS>` works for moves that require two assignments (i.e. `getAssignDelta(`**`var{int}`**` x1, `**`int`**` v1, `**`var{int}`**` x2, `**`int`**` v2)`). Figure A.2 shows the effect of this neighbourhood pictorially. The `MultipleAssignment<LS>`'s ef-

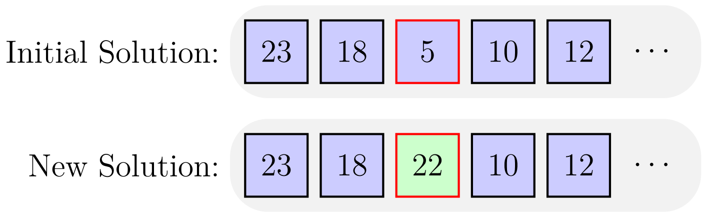


Figure A.1: An example of an `Assignment<LS>` move where variable 3 takes on a new value.

fects could be recreated using two `Assignment<LS>`s. `MultipleAssignment<LS>` does not allow the differentiation of `Function<LS>`s or `DisequationSystem<LS>`s because their interfaces do not provide access to the underlying `getAssignDelta(var{int} x1, int v1, var{int} x2, int v2)` method which `MultipleAssignment<LS>`s wrap around. This does not mean that `Function<LS>`s or `DisequationSystem<LS>`s cannot be used in conjunction with a `MultipleAssignment<LS>` neighbourhood; only that instead of using the efficient differentiation method, it has to fall back to the more costly **lookahead** simulation.

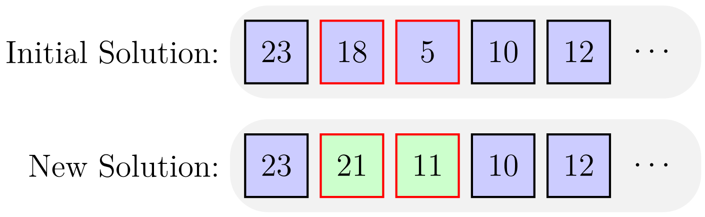


Figure A.2: A `MultipleAssignment<LS>` move where both variables 2 & 3 are altered.

A.1.3 MultipleVariableAssignment<LS>

The `MultipleVariableAssignment<LS>` is an alternate form of the `MultipleAssignment<LS>` that takes two arrays of variables and can perform moves which alter two variables (one from each input array). The `MultipleVariableAssignment<LS>` neighbourhood is designed to operate on models where the two variable arrays are disjoint and distinct yet related by common indices (e.g. in the timetabling model, the time-slots and rooms variables are both indexed by the same events). All neighbours will be at most two atomic changes from the current solution. The maximum number of moves within a `MultipleVariableAssignment<LS>` neighbourhood will be $\left(\sum_{i=0}^{n-1} Domain_{x1_i} + Domain_{x2_i}\right) - 1$. In this case we assume that a move will differ by at least one change but we do not enforce that both variables need to change values.

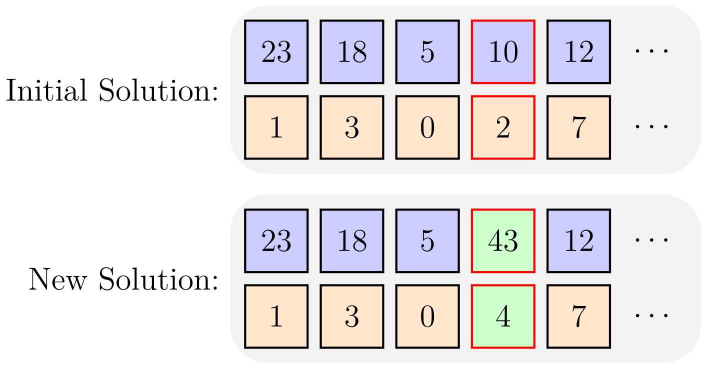


Figure A.3: A `MultipleVariableAssignment<LS>` move changing the values of two variables at index 4.

A.1.4 MultiAssignment<LS>

The `MultiAssignment<LS>` class is the most powerful of the abstract classes because it allows multiple assignments to be performed as one operation. It uses the `getAssignDelta(`**`var{int}`**`[] x, `**`int`**`[] v)` differentiation method. The `MultiAssignment<LS>` class can be used to recreate the effect of any other neighbourhood (even the `Assignment<LS>`). It does not support `Function<LS>`s or `DisequationSystem<LS>`s for the same reasons as `MultipleAssignment<LS>`s and `MultipleVariableAssignment<LS>`s; chiefly, the limited differentiation methods available to non-`Constraint<LS>` classes and the incomplete implementation of these methods for containers other than `ConstraintSystem<LS>`s. Figure A.4 shows a diagram of this move where three variables are being altered. For an array of size n this neighbourhood could alter all n variables. A `MultiAssignment<LS>` could potentially be factorial in size (because the size of the effect is in the range 1 to n).

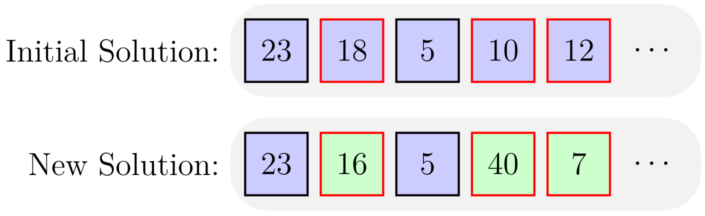


Figure A.4: An example of a three variable `MultiAssignment<LS>` move.

A.1.5 Swap<LS>

`Swap<LS>` is the superclass for any neighbourhood that would ordinarily be differentiated using the `getSwapDelta(`**`var{int}`**` x, `**`var{int}`**` y)` method. Figure A.5 shows a swap operator exchanging the values between the variables at indices 2 & 4. The `getSwapDelta(`**`var{int}`**` x, `**`var{int}`**` y)` method is available for all differentiable objects (probably due to COMET's support for the swap operator, `:=:`). Any neighbourhood implementing `Swap<LS>` will be at most $\binom{n}{2} - 1$ in size and the resulting solutions will be two atomic assignments from the original solution.

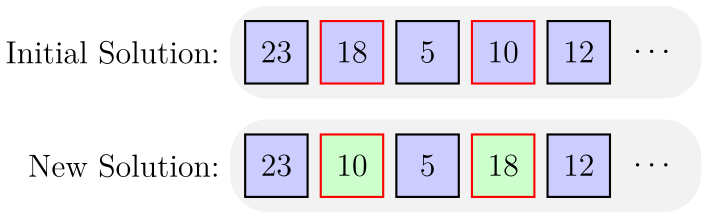


Figure A.5: An example of a `Swap<LS>` between variables 2 & 4.

A.1.6 MultipleVariableSwap<LS>

The `MultipleVariableSwap<LS>` class is to `Swap<LS>` what `MultipleVariableAssignment<LS>` is to `Assignment<LS>`; two swaps performed on two separate yet connected arrays of decision variables. Figure A.6 shows a diagram of this move. This class wraps around the `getSwapDelta(`**`var{int}`**` x1, `**`var{int}`**` y1, `**`var{int}`**` x2, `**`var{int}`**` y2)` method. `Function<LS>`s are not supported; however, `DisequationSystem<LS>`s are. This is because `DisequationSystem<LS>`s can differentiate multiple assignments and a multiple assignment can be used to emulate the effect of a multiple swap; `Function<LS>`s only have single assignment and swap methods and consequently cannot emulate this class. A neighbour from within `MultipleVariableSwap<LS>` will be at most four atomic assignments from the starting solution. As with `Swap<LS>` the neighbourhood could contain up to a maximum of $\binom{n}{2} - 1$ neighbours.

A.1.7 MultipleSwap<LS>

The final abstract neighbourhood is `MultipleSwap<LS>` that, like `MultipleVariableSwap<LS>`, is an extension of the basic `Swap<LS>`. `MultipleSwap<LS>`

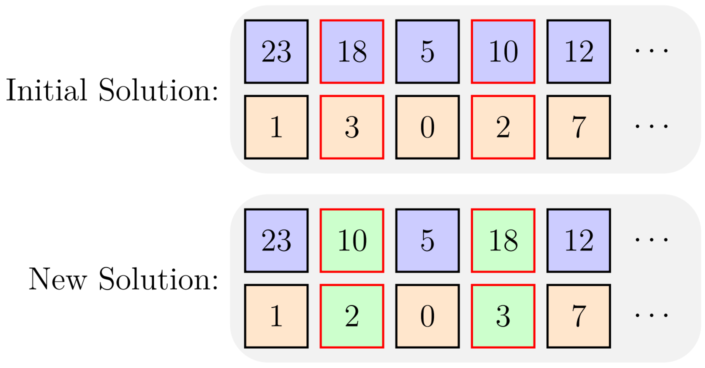


Figure A.6: A `MultipleVariableSwap<LS>` exchanging the values of both arrays at the 2nd and 4th indices.

is actually closer to a `MultipleAssignment<LS>` than a true swap operator. It allows two groups of multiple variables to be assigned two values. The genesis of the operation came from the Timetabling Problem where it is desirable to swap all the events at one time-slot with those at another. So, whilst conceptually the variables are having their values *swapped*, it is actually represented (and differentiated) internally as a multiple assignment. This is also because there is no array parameterised `getSwapDelta` method; only the `getAssignDelta` method has the flexibility to differentiate such a move.

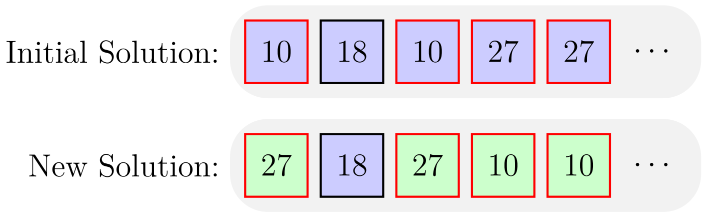


Figure A.7: A `MultipleSwap<LS>` move exchanging the values of two groups. Those with the value 10 and those assigned 27.

A.2 Concrete Neighbourhoods

This section covers the neighbourhoods that provide actual instantiatable implementations of the abstract classes from Section A.1. The concrete neighbourhoods are split into two categories: those that are generic to any problem expressible in Comet and those that rely on information from a particular problem model.

A.2.1 Generic Neighbourhoods

The generic neighbourhoods operate solely on the information available from the COMET variables. They have no dependencies on information or invariants from the problem model. They only require an array of **var{int}**s and a `Model` instance. By supplying different arrays of variables as parameters, at construction time different instances of the same generic neighbourhood classes can behave as completely different neighbourhoods. This can be seen in Table 3.6 where `validRoomAssignments` and `timeslotAssignments` are both `DomainAssignment<LS>`s.

DomainAssignment<LS>

The `DomainAssignment<LS>` class is an implementation of the `Assignment<LS>` class. It operates by selecting values from a variable's domain. Integer decision variables in COMET are associated with domains (just as variables within CSPs have domains). The domain of a **var{int}** is static and does not change during the search process (unless the user explicitly manipulates it); this is different from CP where the domain of a variable is pruned as the search progresses. If the problem being tackled has some static disequations, (e.g. an event cannot be placed in a certain room) then the unwanted values can be removed from the variables' domains prior to the search. This would result in the `DomainAssignment<LS>` neighbourhood only exploring feasible neighbours (with regard to that particular constraint).

SetAssignment<LS>

`SetAssignment<LS>` is another implementation of `Assignment<LS>`. Unlike the `DomainAssignment<LS>` it takes a set of integer values as an argument from amongst which variables must select values. Completely pruning values from a variable's domain may be undesirable; especially if the restrictions in question are from a soft constraint which may not be satisfiable. `SetAssignment<LS>` allows you to supply a separate set to act as a common domain rather than restricting the variables' individual domains. The drawback is that this single domain is common for all variables.

DomainMultipleVariableAssignment<LS>

`DomainMultipleVariableAssignment<LS>` takes two arrays of variables (that share the same indices), selects an index and then reassigns both variables at the chosen index to new values. The values are chosen from amongst the domains of the variables (as was the case for the `DomainAssignment<LS>` class). `DomainMultipleVariableAssignment<LS>` takes the basic operation of the `DomainAssignment<LS>` and by inheriting from `MultipleVariableAssignment<LS>` extends it to operate on two variables. The neighbourhood always causes a change of two assignments and has the size of $\left(\sum_{i=0}^{n-1}(|D_{x1_i}|-1)+(|D_{x2_i}|-1)\right)$

AllSwaps<LS>

The `AllSwaps<LS>` class extends `Swap<LS>` and implements a neighbourhood which tries all the possible exchanges between variables in the supplied array. It restricts the swaps to those where the first selected index is strictly less than the second. This removes symmetric swaps and brings the number of moves down from n^2 to the more manageable $\binom{n}{2}$ (i.e. at least a 50 % saving). Swaps where both variables are currently assigned the same value are also omitted.

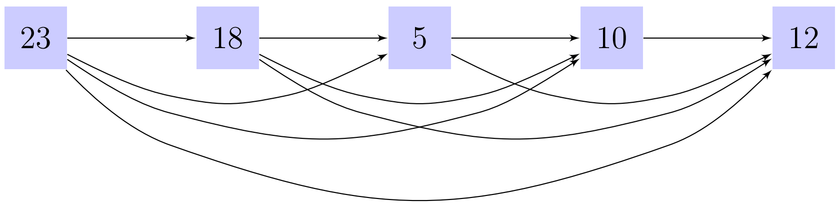


Figure A.8: A diagram showing the values that `AllSwaps<LS>` would consider exchanging.

ConsistentAllSwaps<LS>

`ConsistentAllSwaps<LS>` is another child of `Swap<LS>` by extension from `AllSwaps<LS>`. The *consistency* aspect comes from its use of a second set of variables. The `ConsistentAllSwaps<LS>` limits the swaps to those that share a common value at the same indices in the second array. A simple way of achieving this would be to select the potential swaps exactly as in the `AllSwaps<LS>` and then discard those that do not have identical values in the second array. We choose the more complicated—but efficient—option of using an invariant, `Indices<LS>`. This maintains a mapping between the values of the second array and the indices which are assigned those values. Figure A.9 has a diagram illustrating the two decision variable arrays and the invariant. The values that are not assigned to any variable in B result in

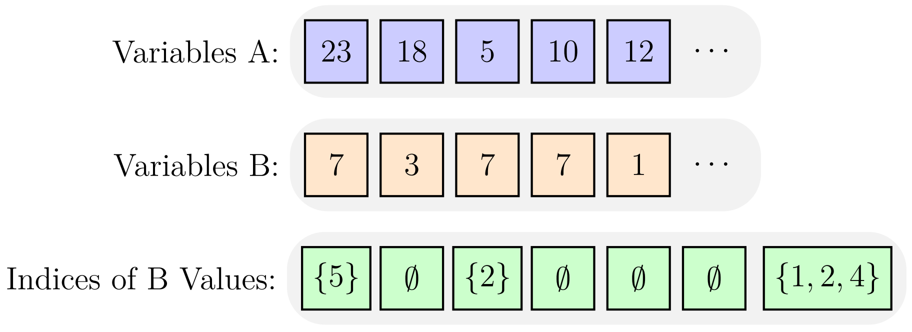


Figure A.9: An example of the pairs of decision variables along with the invariant maintained by `Indices<LS>` class. The domain of Variables B is assumed to be limited to the integers one to seven.

AllMultiVariableSwaps<LS>

`AllMultiVariableSwaps<LS>` is the concrete implementation of the `MultipleV ariableSwap<LS>` class. Like the `AllSwaps<LS>` and `ConsistentAllSwaps<LS>` it chooses two indices from the input array's range and swaps the values; where `AllMultiVariableSwaps<LS>` differs from these neighbourhoods is that it has two arrays of decision variables and exchanges the values in both arrays from the selected indices.

A.2.2 Timetabling Neighbourhoods

The following neighbourhoods have been collected together as *timetabling neighbourhoods* but this does not necessarily mean they are only applicable to timetabling, merely their implementations rely on some information from our timetabling model. Even then the information they need may not actually be problem dependent; for example, the `KempeChain<LS>` requires knowledge about the clashing events but this is supplied as a generic graph (represented as a **set{int}** array) and could just as well be the connected vertices in a K-Colouring Problem. Listing A.1 shows the section of the `OriginalModel` where the relevant invariants are declared. Some of the neighbourhoods perform generic moves that are subject to restrictions derived from the structure of the Timetabling problem (e.g. the number of time-slots in a day, or the number of days). These neighbourhoods are interesting because they demonstrate how the insight and intuition of the algorithm designer can still fit into a decoupled search framework.

```
 roomIndices = new Indices<LS>(eventRooms);

 clashingEvents = p.getClashingEvents();
 eventsStudents = p.getEventsStudents();
 studentsEvents = p.getStudentsEvents();
 timetables = new StudentSchedule(m, eventTimeslots,
studentsEvents, p.getEventsStudents(), times);
 m.post(timetables);

 studentTimetables = timetables.getStudentTimetables();
 if(times.noOfDays > 1){
   studentsEventsPerDay = timetables.getStudentsEventsPerDay()
;
 }

 var{set{int}}[] tI = timeslotIndices.get();
 var{set{int}}[] rI = roomIndices.get();
 actual = new var{set{int}}[r in rooms, t in times.timeslots
](_solver) <- rI[r] inter tI[t];
 clashes = new var{set{int}}[e in events](_solver) <- setof(
f in clashingEvents[e])(eventTimeslots[e] == eventTimeslots[f
]);
 occupancy = new var{int}[r in rooms, t in times.timeslots](
_solver) <- card(actual[r,t]);
 emptyRooms = new var{set{int}}[t in times.timeslots](_solver
) <- setof(r in rooms)(occupancy[r,t] == 0);
 emptyTimes = new var{set{int}}(_solver) <- setof(t in times
.timeslots)(card(emptyRooms[t]) > 0);
 eventsOnDay = new var{set{int}}[d in times.days](_solver) <-
 union(t in times.timeslotsPerDay)(tI[t + (d * times.
noOfTimeslotsPerDay)]);

 var{int} timeCount[t in times.timeslots](_solver) <- sum(r
in rooms)(occupancy[r,t]);
 overfilledRooms = new var{set{int}}[t in times.timeslots](
_solver) <- setof(r in rooms)(occupancy[r,t] > 1);
 overfilledTimes = new var{set{int}}(_solver) <- setof(t in
times.timeslots)(card(overfilledRooms[t]) > 0);
 overfilledTimeslots = new var{set{int}}(_solver) <- filter(
t in overfilledTimes)(timeCount[t] > noOfRooms);
 emptyTimeslots = new var{set{int}}(_solver) <- filter(t in
overfilledTimes)(timeCount[t] == 0);
```

Listing A.1: An excerpt of the `OriginalModel` class (from `OriginalModel.co`) showing the instantiation of the model's invariants (that the neighbourhoods require).

EjectionChain<LS>

The `EjectionChain<LS>` neighbourhood is an implementation of Glover's *Ejection Chain* strategy 1996. The neighbourhood extends the `MultiAssignment<LS>` abstract class as it allows for the reassignment of several events' room and time-slot positions. `EjectionChain<LS>` selects an event and then places at the same position as another event. This second event is chosen and placed at a new location displacing another event. This series of displacements occurs until the chain reaches a specified length; in our experiments this was fixed at three. The final move in the chain tries to choose a location that is empty and will not overlap other events. If this is not possible the event is placed at the same location as another, but no further displacement occurs.

KempeChain<LS>

`KempeChain<LS>` is a version of the Graph Colouring neighbourhood: the Kempe Chain. Kempe Chains are often used in timetabling solvers because of their ability to take a solution without any event clashes and create other solutions that are always feasible. By extending `MultiAssignment<LS>` it can reallocate the time assignments of events occurring at two time periods. A *Kempe Chain* is created by selecting two time-slots and collecting all the events at these times. One event is selected as a starting point. A BFS from the starting vertex across clashing events graph (restricted to just those events currently within the two selected time-slots) builds up the *chain*. Each event visited by the BFS is assigned an alternating time (e.g. the chain is being 2-coloured). Once the chain has been constructed the times can be swapped to generate a new solution. The `KempeChain<LS>` neighbourhood tries all pairs of times and for each pair tries to create a chain starting from every event at those times.

RoomMatching<LS>

`RoomMatching<LS>` differs from the other neighbourhoods in that it is a wrapper around a sub-solver rather than a neighbourhood function in its own right. The sub-solver performs a Bipartite Graph Matching by formulating the problem as a Network Flow. The Push-Relabel algorithm is used to calculate the maximum flow in the graph from which a maximal matching can be extracted. This matching indicates a valid set of room assignments such that all events' domains are respected and there are no overlapping events. `RoomMatching<LS>` is binary in nature; if it finds a matching it will return that to the search as a neighbour. If it finds no matching then no neighbour will be returned. The other neighbourhoods are all capable of returning multiple neighbouring solutions and allowing the search to choose between them. A `RoomMatching<LS>` can be supplied as an argument to a `KempeChain<LS>` neighbourhood. In this situation, after the `KempeChain<LS>` has created a time-slot reassignment it uses the `RoomMatching<LS>` to verify that this this new configuration has a valid room assignment. If no matching can be found the `KempeChain<LS>` discards its move. This particular behaviour was added to emulate the $\mathcal{N}_4(a)$ neighbourhood of Chiarandini et al. [2003, 2006] which performs a Kempe Chain move followed by a bipartite room matching to ensure that feasibility is retained in neighbours.

MoveToEmptySlot<LS>

`MoveToEmptySlot<LS>` is a good example of how invariants can be used to make neighbourhoods more efficient. The neighbourhood selects an event and assigns it to a new *empty* time-slot room position. To create this behaviour one could choose a new time-slot room location for a selected event then check all the other events for any duplication, but this would be tedious and wasteful. Instead we can utilize the `emptyRooms` and `emptyTimeslots` invariants (shown in Listing A.1). Rather than using a trial-and-error approach to finding empty time-slot room pairs, the invariants provide us their exact locations. Using invariants in this fashion means that the `MoveToEventSlot<LS>` neighbourhood's size will be dependent upon the number of empty slots in the particular starting configuration being searched from. The smallest the neighbourhood could be is if there were no overlapping assignments, in which case there would only be the final time-slots empty giving a sizes of $events \cdot (rooms \cdot days)$. In the worst-case scenario all the events have been assigned to a single time-slot room location meaning the neighbourhood would be $events \cdot (time - slotsinaday \cdot rooms \cdot days)$. Other than the dynamic size, the use of the invariants also means it will be guaranteed to have one-way relationship with *overlaps* constraint; a move within this neighbourhood can only ever remove *overlaps* violations, not introduce them.

OverlapRemovingAssignment<LS>

The `OverlapRemovingAssignment<LS>` neighbourhood is a `MultipleVariableA ssignment<LS>` move that expands upon the `MoveToEmptySlot<LS>` basic template. Like the `MoveToEmptySlot<LS>` neighbourhood, `OverlapRemovingAssignment<LS>` places events into time-slot room positions where there are currently no other events. The difference is that `OverlapRemovingAssignment<LS>` only chooses from amongst those events which are in the same time-slot room location as another event. It uses the `overfilledTimes` and `overfilledRooms` invariants from the `OriginalModel` to efficiently focus on the events that need reallocation. It there are no overlapping events in a given configuration then the `OverlapRemovingAssignment<LS>` has no moves to explore.

SwapAllEventsTimeslots<LS>

`SwapAllEventsTimeslots<LS>` is a powerful neighbourhood that extends `Multiple Swap<LS>`. This neighbourhood selected two time-slots (which both contain at least one event) and then exchanges the times for the events. It does not alter the room assignments of the events thus it does not interact with *roomValid* constraint. The second major behavioural property is because the internal composition of the times is *not* altering (e.g. the groups of events at the timeslots are not being manipulated). This means that the neighbourhood does not interact with *eventClashes* or *overlaps*. The `SwapAllEventsTimeslots<LS>` neighbourhood has a strict upper bound of $\binom{timeslots}{2}$ moves (i.e. 990 neighbours in the ITC instances).

SingleDaySwapAllEventsTimeslots<LS>

`SingleDaySwapAllEventsTimeslots<LS>` extends the `SwapAllEventsTimeslots <LS>` neighbourhood. `SwapAllEventsTimeslots<LS>` explores swapping the timeslots of all events at non-empty time-slots. It still interacts with the *singleEvent* constraint because as it transfers events in and out of days it can violate and satisfy the constraint. The `SingleDaySwapAllEventsTimeslots<LS>` restricts the swaps to just between times in the same day. This means that no events are entering or leaving the day they are currently in and consequently any move in this neighbourhood cannot affect the *singleEvent* constraint. This neighbourhood is a good example of where restricting moves to a subset of a more general neighbourhood can enforce specific desirable interaction properties. The `SingleDaySwapAllEventsTimeslots<LS>` neighbourhood has a maximum size of $days \cdot \binom{tPD}{2}$ where tPD is the *timeslots per day*; for the ITC formulation there could be at most 180 neighbours.

ValidRoomAndTimeslotSwaps<LS>

The `ConsistentAllSwaps<LS>` neighbourhood tightly controlled potential swaps; an index could only be swapped with another that it shares a secondary value with (i.e. events exchange times with others assigned to the same room). `ValidRoomAndTimeslotSwaps<LS>` relaxes the restriction that the secondary values need to be the same. It allows swaps to exchange rooms as long as the exchanged values are valid in each other's domains.

SingleDayAllSwaps<LS>

`SingleDayAllSwaps<LS>` is a subset of the general `AllSwaps<LS>` neighbourhood. It selects an event and then only considers exchanging it with events which occur in same day. The implementation uses the `eventsOnDay` **var{set{int}}** (from Listing A.1) to quickly limit the search to only those events which share the same day.

ConsistentSingleDayAllSwaps<LS>

`ConsistentSingleDayAllSwaps<LS>` is another `Swap<LS>` that inherits from `ConsistentAllSwaps<LS>`. It restricts its swaps to those events which are both assigned the same room and occurring in the same day. As with the `SingleDayAllSwaps<LS>`, it uses the `eventsOnDay` invariant but in this case takes the intersection with the room indices (as in the `ConsistentAllSwaps<LS>` neighbourhood).

ConsistentInterDayAllSwaps<LS>

`ConsistentInterDayAllSwaps<LS>` is the complement to `ConsistentSingleDayAllSwaps`; it considers only those swaps that exchange events between separate days. `ConsistentInterDayAllSwaps<LS>` produces a neighbourhood of moves that is the set difference between `ConsistentAllSwaps<LS>` and `ConsistentSingleDayAllSwa`
Another way to look at this relationship is that instead of searching `ConsistentAllSwaps<LS>` the same effects could be achieved by searching both `ConsistentSingleDayAllSwaps<LS>` *and* `ConsistentInterDayAllSwaps<LS>`.

ClashDirectedAssignment<LS>

The `ClashDirectedAssignment<LS>` neighbourhood is an `Assignment<LS>` that uses information about the clashing time-slot assignments to guide its decisions. The neighbourhood uses a combination of two invariants (the time-slot `Indices<LS>` and the `clashes`) and the clashing events sets. The neighbourhood selects an event which conflicts with others at its current time-slot. A new time for the event is selected using the criteria that this new time contains fewer event clashes as the current time.

ClashDirectedAllSwaps<LS>

`ClashDirectedAllSwaps<LS>` extends `Swap<LS>` and expands the basic idea of `ClashDirectedAssignments<LS>` to work for a swap exchanges. An exchange will be allowed if it does not increase the number of clashes in both time-slots.

A.3 Candidate Lists

Candidate Lists were outlined as part of Section 3.3 in Chapter 3; they provide a simple pruning ruleset to reduce the potential neighbours generated by a neighbourhood.

A.3.1 TimesRestrictor<LS>

The only implementation of `CandidateList<LS>` presently in the framework is `TimesRestrictor<LS>`. Amongst the neighbourhoods listed many names share common prefixes and suffixes (e.g. `singleDay`, `MinusFinals`, `InDay`, etc). You may have noticed that in the preceding section there were only nineteen instantiable neighbourhoods listed; less than half the number of neighbourhoods appearing in Chapter 3's interaction detection experiments. This apparent discrepancy can be explained through the use of `TimesRestrictor<LS>`'s. Many neighbourhoods are identical other than the `TimesRestrictor<LS>` that has been supplied to them. The `TimesRestrictor<LS>` has two configurable parameters: `time` and `day`. The time restrictor can be assigned one of two enumerated values, and the day restrictor has three possible values. By varying these arguments it is possible to create six different candidate lists. The `time` argument can be set to:

Valid returns only those time-slots which are not at the end of a day.

All returns all the time-slots.

The `day` choice has three value (two of which use a time-slot as an argument):

None returns all time-slots.

Single takes a time-slot as argument and returns only those slots on the same day.

Different takes a time-slot as an argument and returns those slots on all the other days.

The `TimesRestrictor<LS>`'s rules were only designed to be coherent for the timetabling problem, but the `CandidateList<LS>` interface is generic enough that any ruleset could be captured. Internally, the `TimesRestrictor<LS>` caches the valid sets for a given configuration. The valid sets could be recalculated at each iteration but this would be less efficient.

Appendix B

Additional Data

. . . one of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate the successful termination of their C programs

Dr Robert Firth

This appendix contains data tables or code listings that were too large or disruptive for the main body of the thesis. The first section provides supplementary material from the Graph Colouring experiment in Chapter 3. This experiment formed part of the argument for allowing flexibility with regard to constraint container choice. The second section describes the procedure used to create the starting solutions for each ITC instance and provides a table detailing their associated violations.

B.1 DIMACS data

In Section 3.3 there was an experiment comparing the resource usage of two separate DIMACS Graph Colouring models. Table B.1 provides more detailed results for each instance amongst the data set. The results capture the time to instantiate the model, i.e. the time taken to create the `Solver<LS>` and all the variables, post all the constraints, and then `close` the model. The memory use was recorded once the model had been closed (using COMET's `System.getGCUsage()`

method). The final column represents the average time taken to differentiate a move in the container.

Table B.1: Comparison of the instantiation time T (ms), allocated memory $M(a)$ (MB) and heap memory $M(h)$ (MB) requirements of modelling using a `ConstraintSystem<LS>` versus a `DisequationSystem<LS>` for DIMACS Graph Colouring

			`DisequationSystem<LS>`			`ConstraintSystem<LS>`		
Instance	$\lvert V\rvert$	$\lvert E\rvert$	T (ms)	M (a) (MB)	M (h) (MB)	T (ms)	M (a) (MB)	M (h) (MB)
`anna`	138	986	61	11.38	16	17	8.70	16
`david`	87	812	25	9.28	16	14	8.49	16
`fpsol2.i.1`	496	11654	811	52.53	128	493	27.30	64
`fpsol2.i.2`	451	8691	668	45.28	64	335	23.09	32
`fpsol2.i.3`	425	8688	593	41.58	64	364	23.08	32
`games120`	120	1276	45	10.48	16	20	9.05	16
`homer`	561	3258	1020	64.30	128	58	12.64	32
`huck`	74	602	19	8.78	16	12	8.19	16
`inithx.i.1`	864	18707	2645	140.57	256	915	42.29	64
`inithx.i.2`	645	13979	1439	82.58	128	630	33.40	64
`inithx.i.3`	621	13969	1333	77.69	128	628	33.38	64
`jean`	80	508	21	9.06	16	9	8.17	16
`le450_5a`	450	5714	660	44.56	64	228	18.92	32
`le450_5b`	450	5734	659	44.57	64	254	18.95	32
`le450_5c`	450	9803	727	45.26	64	395	24.56	32
`le450_5d`	450	9757	728	45.25	64	431	24.50	32
`le450_15a`	450	8168	668	44.94	64	300	24.08	32
`le450_15b`	450	8169	669	44.95	64	300	24.09	32
`le450_15c`	450	16680	690	46.36	64	692	38.52	64
`le450_15d`	450	16750	689	46.39	64	740	38.66	64
`le450_25a`	450	8260	671	44.95	64	306	24.29	32
`le450_25b`	450	8263	673	44.96	64	291	24.30	32
`le450_25c`	450	17343	694	46.43	64	717	39.76	64

Continued on next page

Table B.1 - continued from the previous page

			DisequationSystem<LS>			ConstraintSystem<LS>		
Instance	$\|V\|$	$\|E\|$	T	M (a)	M (h)	T	M (a)	M (h)
le450_25d	450	17425	695	46.47	64	711	39.93	64
miles250	128	774	51	10.92	16	16	8.48	16
miles500	128	2340	52	11.08	16	38	11.80	16
miles750	128	4226	63	11.28	16	76	13.29	32
miles1000	128	6432	62	11.51	32	105	15.74	32
miles1500	128	10396	64	11.93	32	215	17.71	32
mulsol.i.1	197	3925	132	15.57	32	128	16.23	32
mulsol.i.2	188	3885	121	14.67	32	129	16.13	32
mulsol.i.3	184	3916	115	14.44	32	127	16.20	32
mulsol.i.4	185	3946	116	14.50	32	130	16.25	32
mulsol.i.5	186	3973	117	14.57	32	126	16.30	32
myciel3	11	20	2	7.76	16	1	7.80	16
myciel4	23	71	2	7.86	16	3	7.89	16
myciel5	47	236	9	8.17	16	8	8.10	16
myciel6	95	755	30	9.59	16	22	9.22	16
myciel7	191	2360	118	14.63	32	81	13.80	32
queen5_5	25	320	3	7.96	16	5	8.02	16
queen6_6	36	580	7	8.03	16	10	8.24	16
queen7_7	49	952	8	8.35	16	16	8.66	16
queen8_8	64	1456	14	8.64	16	20	9.23	16
queen8_12	96	2736	32	9.78	16	50	11.93	32
queen9_9	81	2112	26	9.24	16	32	9.47	16
queen10_10	100	2940	34	9.93	16	50	12.07	32
queen11_11	121	3960	48	10.80	16	67	13.03	32
queen12_12	144	5192	69	12.12	32	86	14.43	32
queen13_13	169	6656	98	13.54	32	103	15.98	32
queen14_14	196	8372	131	15.72	32	143	16.75	32
queen15_15	225	10360	180	17.90	32	225	17.70	32
queen16_16	256	12640	225	21.46	32	237	20.87	32
school1	385	19095	560	37.94	64	878	42.68	64
school1_nsh	352	14612	457	32.74	64	658	34.49	64

Continued on next page

Table B.1 - continued from the previous page

			`DisequationSystem<LS>`			`ConstraintSystem<LS>`		
Instance	$\|V\|$	$\|E\|$	T	$M\ (a)$	$M\ (h)$	T	$M\ (a)$	$M\ (h)$
`zeroin.i.1`	211	4100	161	16.50	32	151	16.01	32
`zeroin.i.2`	211	3541	159	16.42	32	133	15.23	32
`zeroin.i.3`	206	3540	142	16.09	32	128	15.23	32

B.2 ITC Starting Solution Quality

To try to ensure a fair comparison of search enhancements, each run (against a particular instance) was initiated from a fixed starting solution. These solutions were created with a basic constructive algorithm (in Listing B.1) that iterated over all the events and placing them in a unique time-slot / room location. If the selected location is a final time-slot and there are still free locations remaining another selection will be made.

Table B.2 shows each instance from the ITC (and harder) datasets. Each column records the violations reported by the validator for the solution constructed. The final three columns are summations of the appropriate constraints. All the starting solutions were free of *overlap* violations (h_2) and the majority of instances we also free of *finalTimeslot* (s_1) violations.

Table B.2: The quality of the starting solutions for each problem instance.

Instance	h_1	h_2	h_3	s_1	s_2	s_3	H	S	$\forall$
`big_20`	10318	0	902	0	4578	65	11220	4643	15863
`big_19`	9204	0	857	0	3518	84	10061	3602	13663
`big_18`	10704	0	858	0	4591	69	11562	4660	16222
`big_17`	12634	0	785	0	5618	92	13419	5710	19129
`big_16`	6013	0	795	0	2545	234	6808	2779	9587
`big_15`	6733	0	923	0	2555	202	7656	2757	10413
`big_14`	4105	0	917	0	1652	471	5022	2123	7145
`big_13`	4452	0	894	0	1436	433	5346	1869	7215
`big_12`	3978	0	928	0	1302	483	4906	1785	6691
`big_11`	4420	0	1009	1835	1420	408	5429	3663	9092
`big_10`	4128	0	991	1737	1515	369	5119	3621	8740
`big_9`	3190	0	955	1351	1236	268	4145	2855	7000

big_8	3989	0	952	1835	1615	391	4941	3841	8782
big_7	6965	0	1004	2235	2482	215	7969	4932	12901
big_6	5092	0	1015	1992	1923	246	6107	4161	10268
big_5	4079	0	1003	1919	1512	352	5082	3783	8865
big_4	3540	0	956	1429	1225	327	4496	2981	7477
big_3	3255	0	925	0	1133	473	4180	1606	5786
big_2	3429	0	924	0	1446	515	4353	1961	6314
big_1	2991	0	941	0	930	698	3932	1628	5560
med_20	8931	0	246	2285	4176	81	9177	6542	15719
med_19	8434	0	219	2577	4328	36	8653	6941	15594
med_18	9854	0	208	0	5019	86	10062	5105	15167
med_17	4684	0	252	0	2309	135	4936	2444	7380
med_16	8816	0	198	0	4148	104	9014	4252	13266
med_15	2126	0	238	955	841	194	2364	1990	4354
med_14	3073	0	203	0	1562	569	3276	2131	5407
med_13	4847	0	246	0	1624	198	5093	1822	6915
med_12	3079	0	254	0	1274	276	3333	1550	4883
med_11	2725	0	216	0	1040	491	2941	1531	4472
med_10	1680	0	205	0	593	281	1885	874	2759
med_9	3385	0	347	0	1546	38	3732	1584	5316
med_8	2040	0	318	0	1016	92	2358	1108	3466
med_7	2975	0	359	0	1446	46	3334	1492	4826
med_6	2147	0	357	0	903	118	2504	1021	3525
med_5	1815	0	335	821	789	155	2150	1765	3915
med_4	1508	0	333	733	727	165	1841	1625	3466
med_3	1732	0	321	0	728	161	2053	889	2942
med_2	1310	0	322	0	512	221	1632	733	2365
med_1	1336	0	343	0	478	215	1679	693	2372
small_20	2767	0	114	1890	1545	912	2881	4347	7228
small_19	7289	0	86	2433	3660	72	7375	6165	13540
small_18	5384	0	116	2348	2313	214	5500	4875	10375
small_17	8100	0	84	0	4633	55	8184	4688	12872
small_16	3525	0	78	0	2440	232	3603	2672	6275
small_15	3796	0	86	0	1831	248	3882	2079	5961
small_14	3213	0	68	1872	1539	235	3281	3646	6927
small_13	3287	0	141	1887	1514	284	3428	3685	7113
small_12	1675	0	168	1207	415	1133	1843	2755	4598
small_11	3149	0	125	0	1101	698	3274	1799	5073

small_10	5281	0	148	2053	2730	176	5429	4959	10388
small_9	4814	0	109	1964	2943	107	4923	5014	9937
small_8	3665	0	146	1811	1591	276	3811	3678	7489
small_7	2829	0	123	0	1198	386	2952	1584	4536
small_6	1360	0	94	0	662	1112	1454	1774	3228
small_5	1934	0	110	0	1028	175	2044	1203	3247
small_4	1604	0	104	0	675	191	1708	866	2574
small_3	2711	0	86	0	1225	73	2797	1298	4095
small_2	1310	0	113	0	606	248	1423	854	2277
small_1	599	0	94	0	259	124	693	383	1076
competition20	1134	0	237	0	382	190	1371	572	1943
competition19	987	0	234	0	369	153	1221	522	1743
competition18	642	0	334	0	255	103	976	358	1334
competition17	1016	0	306	0	392	169	1322	561	1883
competition16	811	0	313	0	303	105	1124	408	1532
competition15	950	0	274	0	368	134	1224	502	1726
competition14	1071	0	241	0	465	193	1312	658	1970
competition13	809	0	308	0	331	132	1117	463	1580
competition12	647	0	321	0	246	92	968	338	1306
competition11	750	0	313	0	286	126	1063	412	1475
competition10	694	0	268	0	270	102	962	372	1334
competition09	674	0	333	0	270	121	1007	391	1398
competition08	844	0	270	0	354	107	1114	461	1575
competition07	965	0	244	0	559	190	1209	749	1958
competition06	1104	0	232	0	356	131	1336	487	1823
competition05	1036	0	280	0	387	160	1316	547	1863
competition04	986	0	305	0	430	153	1291	583	1874
competition03	655	0	259	0	225	104	914	329	1243
competition02	564	0	329	0	270	104	893	374	1267
competition01	714	0	324	0	299	108	1038	407	1445

```
  void initialisePermutationSolution(){
    range slots = times.timeslots.getLow()..(noOfRooms * times.
   noOfDays * (times.noOfTimeslotsPerDay))-1;
    RandomPermutation P(slots);
    int resets = 0;
    with atomic(_solver){
      forall(e in events){
        int location = P.get();
        while((location / noOfRooms) % times.noOfTimeslotsPerDay
   == times.noOfTimeslotsPerDay - 1 && slots.getSize() - resets >
    events.getSize()){
          location = P.get();
          resets++;
        }

        eventTimeslots[e] := location / noOfRooms;
        eventRooms[e] := location % noOfRooms;
      }
    }
  }
```

Listing B.1: The solution construction routine from OriginalModel.co.

Glossary

Ant Colony Optimisation (ACO) a constructive search technique which is inspired by the way ants use pheromone trails to find food sources.

Adjacent indicates that two vertices are connected by an edge or arc.

Artificial Intelligence an area of Computer Science which attempts to use computers to solve problems using intelligent processes.

Adaptive Iterated Construction Search (AICS) a family of constructive Local Search strategies which includes GRASP, SWO and ACO.

Arc a directed edge between two vertices found in digraphs.

Breadth-First Search (BFS) a complete tree search algorithm which works by exploring all the nodes at one level before expanding to the next layer of the tree.

Bin Packing Problem (BPP) an optimisation problem where items of varying sizes must be assigned to as few fixed-capacity containers as possible.

Constraint-Based Local Search (CBLS) a movement in the Local Search community towards a CP style declarative modelling of problems in terms of constraints, allowing for more loosely coupled algorithms.

Constraint-Directed Neighbourhoods (CDN) a technique by Ågren [2007] which generates neighbourhoods from a set based model of a problem's constraints.

Constraint Directed VNS (CDVNS) a form of VNS proposed by us.

Conjunctive Normal Form (CNF) a series of clauses containing the disjunction of literals that are joined by conjunctions, e.g. $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_4)$.

Constraint-Oriented Neighborhoods (CON) a technique introduced by Viana et al. [2005] which uses specialised neighbourhoods to achieve moves which respect certain problem constraints.

Constraint Optimisation Problem (COP) similar to CSPs with additional soft constraints that form an objective function to be optimised.

Constraint Programming (CP) a search technique which uses constraints to guide a backtracking DFS search combined with propagation and consistency algorithms to filter out inconsistent values.

Constraint Satisfaction Problem (CSP) a problem where a set of variables must be assigned values from domains which respect the limits imposed by a set of constraints on their feasible values.

Directed Acyclic Graph (DAG) a restricted form of directed graph with no cycles.

Depth-First Search (DFS) a complete tree search which explores all nodes by delving to the deepest node and then recursing back visiting the next deepest unexplored node.

The Center for Discrete Mathematics and Theoretical Computer Science (DI
a collaborative academic and industrial research institute based at Rutgers University in New Jersey. It has sponsored various *challenges* to focus algorithm research on specific open problems.

Dynamic Local Search (DLS) a metaheuristic strategy which applies penalties to solution components, the most widespread example is the GLS algorithm of Voudouris and Tsang [1995].

Digital Object Identifier (DOI) a system for uniquely (and persistently) identifying content on a digital network.

Davis Putnam Logemann Loveland (DPLL) a complete backtracking search algorithm used for solving SAT problems which was introduced by (and named after) Davis et al. [1962].

Edge a connection between two vertices in a graph.

Fitness-Distance Correlation (FDC) captures the notion that solutions with high fitness should be close to the optimal solution.

First Fit Decreasing (FFD) an approximation algorithm for the BPP.

Finite State Machine (FSM) an abstract mathematical model used to represent the behaviour of a stateful logic system.

Genetic Algorithm (GA) a form of search algorithm inspired by natural evolution. Solutions are encoded as strings of characters which can be combined with other solutions or mutated to provide new generations of solutions.

Generalised Arc Consistency (GAC) a form of domain consistency where for every value within a variables' domain there exists values in the other variables' domains which satisfy the constraint.

Great Deluge Algorithm (GDA) a metaheuristic by Dueck [1993] which behaves like a simplified SA.

Guided Local Search (GLS) a metaheuristic which operates by manipulating the acceptance function of the search.

Generalised Local Search Machine (GLSM) a FSM model of Local Search algorithms to formalise their study by Hoos and Stützle [2005, Chp. 3, p. 113].

Genetic Programming (GP) an AI strategy where (rather than directly finding a solution to a problem) programs are evolved to solve the task.

Greedy Randomised Adaptive Search Procedures (GRASP) a form of stochastic, constructive Local Search introduced by Feo and Resende [1989].

Iterative Deepening Depth-First Search (IDDFS) a variant of DFS which imposes a limit for the search depth. If no solution is found then this limit is increased and the search is restarted. An upper bound is usually placed on the limit's potential expansion.

Iterated Local Search (ILS) a metaheuristic which couples repeated Local Search runs with perturbation to escape from local optima.

Integer Programming (IP) a form of LP in which variables can be restricted to solely integer values.

International Timetabling Competition (ITC) a series of competitions held to unify research into automated timetabling.

Large Neighbourhood Search (LNS) a Local Search strategy introduced by Shaw [1998] which uses a CP search as a neighbourhood.

Local Search a term referring to a class of iterative improvement neighbourhood search algorithms.

Linear Programming (LP) an optimisation tool where problems are formulated as linear equations and solved using mathematical techniques.

Mixed Integer Programming (MIP) a form of LP and IP which can model expressions containing both integer and real variables.

Neighbour a member of a neighbourhood.

Neighbourhood in *Graph Theory* the set of vertices that are adjacent to the current vertex. In the context of Local Search a neighbourhood is the function which permutes the current solution to create the new solutions.

Object-Oriented Programming (OOP) a programming paradigm which uses the concept of *objects* to encapsulate data (and methods which manipulate it) in a single place.

Optimization Programming Language (OPL) a language for expressing CP and mathematical programming models by Van Hentenryck [1999].

Operations Research (OR) a branch of applied mathematics focused on solving practical optimisation problems.

Probabilistically Approximately Complete (PAC) due to sufficient randomness in the search the probability of exploring all solutions tends to 1 as the running time increases towards infinity.

Portable Document Format (PDF) an open standard for documents originally developed by Adobe Systems. Usually referred to by its file extension, .pdf.

Progressive Party Problem (PPP) an assignment problem based around organising a social gathering at a yacht club.

Simulated Annealing (SA) a heuristic which escapes local optima by accepting non-improving states with a probability that is steadily decreased as the search progresses.

Boolean Satisfiability (SAT) a classic $\mathcal{NP}$-complete problem in which a series of boolean variables must be assigned values to cause the conjunction of clauses containing these variables (in CNF) to become true.

Strongly Connected Component (SCC) a directed graph where there is a path from each Vertex to every other.

Set a mathematical collection of distinct objects usually denoted using curly braces e.g. $\{1, 2, 3, 4\}$, $\{red, blue, green\}$.

Squeaky Wheel Optimization (SWO) a greedy constructive search which identifies constrained variables and then adjusts their priority for earlier assignment in subsequent construction iterations.

Tabu Search (TS) a metaheuristic which has a short-term memory of previously explored states to try to avoid wastefully revisiting the same solutions.

Travelling-Salesman Problem (TSP) a classic optimisation problem where the objective is to complete a Hamiltonian tour of cities in the shortest distance possible.

Unit Commitment Problem (UCP) a problem concerned with satisfying a forecasted energy demand by switching on and off various power generation systems over a time horizon.

Unified Modelling Language (UML) a general-purpose language for capturing software system designs.

Uniform Resource Locator (URL) a system for addressing locations, chiefly known as the way sites on the internet are described.

Variable Depth Search (VDS) a search technique where search moves are composed by applying several steps from different neighbourhoods to create one single larger move.

Vertex a node within a graph.

Variable Neighbourhood Descent (VND) the most prevalent VNS strategy whereby neighbourhoods are explored in order of expanding size until an improvement is found at which point the search returns to the first neighbourhood.

Variable Neighbourhood Search (VNS) a Local Search strategy which alters the neighbourhoods in a systematic fashion. It was first codified by Mladenović and Hansen [1997].

Vehicle Routing Problem (VRP) a problem similar in nature to the TSP in which a series of deliveries must be made using a finite fleet of vehicles.