APPLYING SYSTEMATIC LOCAL SEARCH TO
JOB SHOP SCHEDULING PROBLEMS

by

Lei Duan
M.Sc., Essex University, England, 2003
B.Eng., Tsinghua University, Beijing, China, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Lei Duan 2006
SIMON FRASER UNIVERSITY
Summer 2006

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

Name: Lei Duan
Degree: Master of Science
Title of thesis: Applying Systematic Local Search to Job Shop Scheduling Problems

Examing Committee: Dr. Robert F. Hadley
Chair

Dr. William S. Havens, Senior Supervisor

Dr. Fred Popowich, Supervisor

Dr. Uwe Glässer, SFU Examiner

Date Approved: 11 JULY 2006
DECLARATION OF
PARTIAL COPYRIGHT LICENCE

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

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

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

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

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

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

Simon Fraser University Library
Burnaby, BC, Canada
Abstract

In this thesis, an instance of Systematic Local Search, a hybrid search method developed previously, is defined for the job shop scheduling problem. In particular, a nogood is defined as a set of variable assignments on the precedence relations of the critical path. A nogood is a set of variable assignments that is precluded from any solution to the problem. We evaluate the effectiveness of this instance on benchmark job shop scheduling problems. Experimental results show that Systematic Local Search outperforms conventional heuristic search methods such as simulated annealing and tabu search and compares favourably with methods designed specifically for job shop scheduling problems. We also analyze the algorithm’s runtime performance as well as the nogood utilization during the search.
To my family
How wonderful to be wise, to understand things, to be able to analyze them and interpret them. Wisdom lights up a man’s face, softening its hardness.

— Ecclesiastes 8:1, BIBLE
Acknowledgments

First and foremost, I would like to thank most sincerely my senior supervisor Dr. Bill Havens for his professional supervision and kind support during the past three years. Although it took a bit longer than I expected to complete a research MSc degree, Dr. Havens's generous advice, help and patience gave me strength to finish the degree work and become more mature as a researcher. Without all his attention, completing this degree would have not been possible.

I also wish to thank my supervisor Dr. Fred Popowich for his kind advice and help on my thesis work. He is always an interesting and benign professor and helped very generously with this thesis work.

Many thanks to Dr. Uwe Glässer for being my thesis examiner and part of my thesis committee. His kind advice and comments greatly benefited this thesis.

I would also like to pay tribute to Bistra Dilkina, who helped greatly with understanding the original work which inspired this thesis research. I always find discussion with her thought-provoking. I also thank Morten Irgens and Junas Adhikary at Actenum Corporation who granted me the privilege of using and testing my algorithm on ConstraintWorks.

My gratitude also goes out to my colleagues who worked with me in the laboratory. Their discussions and assistance made this research experience truly enjoyable and rewarding.
# Contents

Approval ii  
Abstract iii  
Dedication iv  
Quotation v  
Acknowledgments vi  
Contents vii  
List of Tables ix  
List of Figures x  

1 Introduction 1  
   1.1 Search Methodologies .................................. 2  
   1.2 Search Applications .................................. 4  
   1.3 Organization of the Thesis ................................. 5  

2 Preliminaries 6  
   2.1 Constraint Satisfaction Problem ............................. 6  
   2.2 Heuristic Evaluation ........................................ 10  

3 Systematic Local Search for Optimization 13  
   3.1 Underlying ideas ........................................... 13  
   3.2 The algorithm ............................................. 15
List of Tables

5.1 Experimental results of SysLS on problem instances la01–40 .............. 30
5.2 Experimental results of SysLS for problem instances ft06, ft10, ft20, orb01–10 and abz5–9 (18 instances in total) ......................... 31
5.3 Comparison on the Lawrence problem set (40 instances). OPT is the number of instances which are solved to optimality. MRE is the mean relative error of the best makespan found. .................................................. 32
5.4 A comparison of SysLS with other heuristic approaches on the best makespan found for a total of 11 very hard JSSP instances. The notation “—” means no result is available for the selected instance. The MRE is the mean relative error over the selected 11 instances. ............................................. 33
List of Figures

2.1 A simple example of a Constraint Satisfaction Problem 7

3.1 SysLS for constraint optimization problems from [10] 15
3.2 An illustration of state transition in Systematic Local Search 17

4.1 An example of a 3-job, 3-machine instance of a job shop scheduling problem 20

5.1 Box plot for the aggregated run-time performance on la21 over 10 runs. The maximum number of iterations is set to 2000. 34
5.2 Box plot for the aggregated run-time performance on orb04 over 10 runs. The maximum number of iterations is set to 2000. 35
5.3 Iterations at which all the nogoods are induced versus iterations at which some of the nogoods are in the way of the search for a single run of the SysLS for instance la01 37
5.4 Progress on the makespan versus number of iterations for instance la01 38
Chapter 1

Introduction

The Constraint Satisfaction Problem (CSP) [38, 34] provides a very powerful tool for modelling a variety of problems in Artificial Intelligence (AI). Although the problem itself can exhibit a complex nature during its solving, the modelling can be as simple as the specification of a set of variables whose values are drawn from a finite domain with a set of constraints restricting the values that the variables can assume in a solution to the problem. In fact, with the addition of an optimization criterion which is usually expressed as an objective function, the CSP can be also used to conveniently model many optimization problems in which some solutions are better than others. Since its introduction in AI, the CSP research community has made substantial progress on problem solving. Some of the main solving methods include, but are not limited to, problem reduction measures [38] such as enforcing consistency and breaking symmetry, search algorithms [34] such as chronological backtracking, intelligent dynamic backtracking and stochastic local search, and complexity analysis techniques [17] such as phase transition [41]. These methods have been applied to many prevalent real-world applications in transportation, telecommunication, resource management and even bioinformatics and achieved significant success, making CSP a widely recognized method. Recent advances in distributed constraint satisfaction [42] have also paved the way for its applications in robotics and multi-agent systems.

In spite of all these exciting developments and successes, CSP has still yet to realize its full potential. From our point of view, researchers focused on CSP face two imminent challenges. The first is to make its solving methods, especially the search methods, more effective and the second is to make its application more prevalent. We will examine these two challenges in turn before we position our research contributions in this thesis. Note
that we shall first limit our attention to satisfaction problems without optimization criteria, in which context CSP exhibits many of its original characteristics. Later we shall relax this restriction as we formally include definitions for optimization of the CSP model in Chapter 3. We will also outline the structure of the rest of the thesis at the end of this chapter.

1.1 Search Methodologies

Solving a CSP relies heavily on search algorithms. Since many problems that we are trying to deal with are intrinsically intractable, it is impractical to enumerate all the possible assignments of values to variables in order to find all the solutions, nor to enumerate all the possible solutions to find the best one for optimization problems no matter what powerful computing resources we can get. This means that a constructive search method, which is based on a tree structure and tries to progressively extend to a full assignment, can be disastrous when a mistake is made early in the search tree because the mistake is going to persist for an exponentially long time before backtracking may possibly correct it. Although the constructive search method is meant to be complete, meaning that it is guaranteed to find a solution if there is one and is able to prove no solution exists if there is none, it is very difficult to apply it to problems of even moderate size to get satisfactory performance and its performance further deteriorates when the size of the problem gets bigger. So far no known technique can entirely prevent these early mistakes from being made at or close to the root of the search tree where heuristic information about the search space is least available.

In contrast, stochastic local search methods [19] are immune to the pitfall of early mistakes. Typically local search operates on a complete assignment of values to variables and follows a heuristic evaluation function (typically the objective function itself) to guide the search. As a result, the completeness of the search is usually abandoned, as for example in tabu search [15]. Stochastic local search with full assignment is obviously much more informative than constructive search where the search is working on partial assignments most of the time. However the stochastic nature of local search makes remembering search history problematic. Since most local search methods are equipped with either very limited memory (e.g. tabu search) or virtually no memory at all, it is not possible for them to systematically cover the search space. Therefore a local search method can neither guarantee to find a solution if the problem is feasible nor to prove no solution exists if the problem is
infeasible. Nevertheless local search usually performs quite well in practice, especially when the size of the problem becomes large.

In the past decade or so, people in CSP research have become increasingly interested in hybrid search methods which synthesize desirable aspects of both constructive search and local search. There are generally two ways of hybridizing constructive search and local search. One is to introduce stochastic local search into a basically constructive search framework. For example, local search on full or nearly-full assignment is performed at or close to the leaf of a search tree. In such a case, local search is used with constraint propagation and "look-ahead" techniques for pruning the search tree to optimize the results obtained by a constructive search. Then the constructive search is called again to initiate another iteration until a solution (i.e. full assignment) is reached or a specified number of iterations has elapsed [35]. As a result, completeness is forsaken in exchange for the flexibility to move heuristically in the search space. The other approach is to incorporate constructive search within a stochastic method. The key to systematically covering the search space is to induce a nogood (a set of variable assignments that is precluded from any feasible solution to the problem) at every step of the search [13]. Our method, Systematic Local Search (SysLS) follows this approach. First reported in [18], SysLS searches through a space of complete variable assignments to find a feasible solution. The concept of a maximally consistent solution relaxes the requirement for maintaining feasibility as in constructive search. It avoids the shortcoming of constructive search where the most persistent decisions are made with the least heuristic information while preserving full freedom to move in the search space with a heuristic variable ordering. On the other hand, many local search methods easily get trapped in local optima. SysLS overcomes local optima by recording them as nogoods in a search memory. SysLS treats nogoods as constraints without choosing a particular variable as a culprit. Nogoods force variables away from maximal but unacceptable solutions to the CSP and also enable the search to systematically cover the search space. We will describe SysLS in greater detail in Chapter 2.

Next we are going to briefly discuss some applications, especially the resource scheduling application, to which we are going to apply the Systematic Local Search.
1.2 Search Applications

Resource scheduling pertains to scenarios in which there are multiple tasks demanding a scarce resource; precedence relations between tasks must be imposed in order to maximize the resource's utility. Resource scheduling can be found in many challenging combinatorial optimization problems and has numerous important applications in transportation, telecommunication, networking and logistics. The solution involves imposing temporal relations between tasks to be processed on different resources subject to a variety of side constraints such as resource capacity, task precedence, deadlines and time windows. Although resource scheduling has attracted much research attention and has been repetitively attempted by different methods, it still remains very challenging [36]. Even a simplified version with very few side constraints, such as job shop scheduling, is notoriously difficult to solve optimally. The complex structure shared by resource scheduling problems in general makes them even harder to solve when they appear in real-world applications such as satellite scheduling [14], where the resource (the satellite) becomes so expensive and precious that its utilization must be carefully optimized. When it comes to solving methods, heuristic search is very effective and widely used in solving scheduling problems and building tools for scheduling applications (e.g. ILOG Scheduler\(^1\)). Both constructive search and stochastic local search have been used. Constructive search usually starts off with an empty schedule and tries to add tasks into the schedule one by one. It backtracks if a constraint is violated or the objective function value turns out to be less than optimal. Constraint propagation, which takes advantage of the problem structure, is enforced to prune the search space and to help find solutions quickly [39]. However the problem associated with constructive search for scheduling is that the search algorithm does not scale well with the problem size. On the other hand, local search can be applied when the size of the problem becomes large or the complexity of the problem makes constructive search less effective. Local search starts from a full schedule which is produced randomly or by some constructive heuristic method. Then it tries to iteratively repair the schedule by simple moves in the neighbourhood such as swapping two adjacent tasks [25]. During the search process, the objective function value does not change monotonically but rather gets progressively optimized as in [6]. Also local search allows the schedule to be incrementally updated [23], thus offering great efficiency when evaluating and updating solutions. However local search usually gets stuck in local

\(^1\)http://www.ilog.com/products/scheduler/
optima and the lack of memory makes local search less effective in getting out of local optima and exploring a bigger part of the search space. Due to this deficiency, local search sometimes gives rather poor results [2].

More powerful search methods can make CSP more prevalent in real-world applications. The objective of designing Systematic Local Search and in particular extending it for optimization problems is to come up with a powerful search method synthesizing desirable aspects of both constructive search and stochastic local search for solving challenging combinatorial problems. In the meantime, the search method is flexible enough to accommodate different heuristics based upon the structure of specific problems. As we will describe in greater detail in the following, we will show that the original framework of Systematic Local Search can work quite well for optimization problems by treating each local optimum as a nogood violating a constraint corresponding to the optimization criterion. Therefore all the definitions in the Systematic Local Search for CSP can be harmoniously extended or redefined for optimization. In this way both constraint and optimization criteria can be treated as nogoods thus unifying the treatment of each. It should then be easier to experiment with specific heuristics for repairing nogoods in order to satisfy constraints as well as to optimize. We shall introduce all the necessary definitions, algorithms and examples in order to demonstrate the effectiveness of Systematic Local Search.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 will present the original Systematic Local Search described in [18] with all the necessary definitions and background. Chapter 3 will describe how to extend it for optimization problems and will present the algorithm which works for both CSP and optimization problems. Chapter 4 will provide a case study of the application of SysLS to the job shop scheduling problem and will discuss how to implement the algorithm for efficiency and good performance. Chapter 5 will present experimental results for benchmark job shop scheduling problems. Chapter 6 will conclude the thesis with a vision for future work.
Chapter 2

Preliminaries

We begin this chapter with a formal description of the seminal work on Systematic Local Search [18], designed originally for solving constraint satisfaction problems. We will also look ahead to its extension for optimization problems where appropriate.

2.1 Constraint Satisfaction Problem

First of all, let us take a look at the definition of the Constraint Satisfaction Problem followed by a very simple example.

Definition 2.1. A Constraint Satisfaction Problem (CSP) is represented as a tuple \((V, D, C)\) where \(V\) is a set of variables that can take values from their domain \(D\), and \(C\) is a set of \(k\)-ary constraints \((Ic)^k\) which specify the allowed assignments of the set of \(k\) variables.

In Figure 2.1, four variables \((V_1, V_2, V_3, V_4)\) are represented as vertices and their inequality constraints are represented as edges. For example, there is an edge connecting \(V_1\) and \(V_2\) which corresponds to the constraint \(V_1 \neq V_2\). The corresponding CSP can be modelled by \((V, D, C)\) where \(V = \{V_1, V_2, V_3, V_4\}\), \(D = \{1, 2, 3\}\) and \(C = \{V_1 \neq V_2, V_1 \neq V_3, V_2 \neq V_3, V_2 \neq V_4, V_3 \neq V_4\}\).

A solution to a CSP is a complete assignment of all variables that satisfies all the constraints (also known as a consistent assignment). Note that if we represent the constraint \(C\) as a set of “forbidden” variable assignments which are not part of any solution to the problem, as represented in [13], then a solution to the problem is a set of complete variable assignments that is disjoint from \(C\). It is easy to figure out that a solution to the problem in
our example (Figure 2.1) is \( V_1 = 1, V_2 = 2, V_3 = 3, V_4 = 1 \) which satisfies all the constraints of \( C \).

Note that for each variable \( x \in V \), the size of its domain \( D_x \) can be different. The size of the CSP is generally \( \prod_{x \in V} |D_x| \). Therefore reducing the size of the domain by carefully modelling the problem can contribute to a reduction of the overall problem size. In fact many problem reduction methods in CSP work to reduce the domain size either by eliminating those values that are incompatible with each other and can never appear in a solution together, for instance arc-consistency [33], or by ignoring those domain values that are identical to each other due to symmetry [7, 32].

During search, a variable assignment acts as a unary constraint imposed on the variable in question and may be constantly modified. This type of constraint is referred to as a decision constraint as in [22]. In constructive search method, a series of such decision constraints may make a particular variable assignment impossible since the domain becomes empty due to constraint propagation. We can take the view that all these decision constraints collectively constitute an explanation for the failed variable assignment. To continue with the search, some decisions must be changed. We can use chronological backtracking to alter
the latest decision or some heuristic method to change the most plausible decision. In order to distinguish variable assignments at different stages of the search, we will, from here onwards, refer to the value assignment of a variable imposed by the search at the current time as the current assignment of this variable and the set of all current assignments as the current solution. A solution can contain assignments for all the variables as in local search or assignments for a subset of all the variables as is typical in constructive search. We define such a set of variable assignments as a label.

**Definition 2.2.** A label \( \lambda(X) = \{(x = a) \mid x \in V \land a \in D_x\} \) is a set of variable assignments where \( X \subseteq V \) and \( a \in D_x \). [18]

**Definition 2.3.** A label \( \lambda \) is valid if and only if every variable assignment \( (x = a) \in \lambda \) is the current assignment of the variable \( x \). [18]

A label may not contain all the variable assignments. In the simple example we introduced above, \( \{(V_1 = 1), (V_2 = 2)\} \) is a label which does not contain variable assignments for \( V_3 \) and \( V_4 \). At this point, we can introduce the concept of a nogood, which is a label specifying known inconsistent variable assignments.

**Definition 2.4.** A nogood is a label \( \lambda_\bot = \{(x = a)\}_{x \in X}, X \subseteq V \) such that no solution to the CSP contains the variable assignments of \( \lambda_\bot \). [18]

A simple example of a nogood is \( \{(V_1 = 1), (V_2 = 2), (V_3 = 1), (V_4 = 3)\} \) which violates the constraint \( V_1 \neq V_3 \). Similarly, \( \{(V_1 = 1), (V_2 = 2), (V_3 = 2), (V_4 = 3)\} \) violates the constraint \( V_2 \neq V_3 \) and \( \{(V_1 = 1), (V_2 = 2), (V_3 = 3), (V_4 = 3)\} \) violates \( V_3 \neq V_4 \). All the nogoods are stored in a nogood memory \( \Gamma \). In order to be more precise about an allowed or disallowed assignment with respect to the current variable assignments, we add the following definition.

**Definition 2.5.** An assignment \( (x = a) \) is disallowed if and only if \( \exists \lambda_\bot \in \Gamma, (x = a) \in \lambda_\bot, \) and \( \lambda_\bot \setminus \{(x = a)\} \) is valid. Otherwise the assignment is allowed. [18]

Clearly a variable assignment is disallowed if it “triggers” a nogood stored in \( \Gamma \) to appear in the current solution. For example \( (V_3 = 1) \) is disallowed given \( \lambda_\bot = \{(V_1 = 1), (V_2 = 2), (V_3 = 1), (V_4 = 3)\} \in \Gamma \) and so is \( (V_1 = 1), (V_2 = 2) \) and \( (V_4 = 3) \). We can now have a better understanding of the functionality of nogoods in a search. They help a search remember those regions in the search space represented by labels that contain no solution.
to the problem, hence preventing the search from visiting these parts of the search space. They also guide the search towards promising regions where a solution may exist. Therefore we will be only interested in these allowed assignments.

The set of all the allowed assignments of a variable \( x \) defines the live domain of \( x \).

**Definition 2.6.** The **live domain** \( \Delta_x \) of a variable \( x \) is all of the allowed values from its domain, i.e. \( \Delta_x = \{ a \in D_x | (x = a) \text{ is allowed} \} \). [18]

When the live domain of a variable becomes empty, there exists some nogoods in the nogood store \( \Gamma \) disallowing every domain element. In the above example, suppose that \( \Gamma \) contains three nogoods: \( \{(V_1 = 1), (V_2 = 2), (V_3 = 1), (V_4 = 3)\} \), \( \{(V_1 = 1), (V_2 = 2), (V_3 = 2), (V_4 = 3)\} \), and \( \{(V_1 = 1), (V_2 = 2), (V_3 = 3), (V_4 = 3)\} \). These nogoods disallow every domain element for variable \( V_3 \), hence the live domain of \( V_3 \) becomes empty. In such a case, a **nogood resolution** allows the inference of a new nogood from a set of known nogoods. The nogood resolution can be somewhat loosely described as follows.

**Definition 2.7.** (Nogood Resolution Rule) Provided that for a variable \( x \) with its domain \( D_x = \{a_1, a_2, \ldots, a_d\} \), we have valid nogoods in \( \Gamma \) such that \( \lambda_1^1 = \{ (x = a_1) \wedge N_1 \} \), \( \lambda_1^2 = \{ (x = a_2) \wedge N_2 \} \), \ldots, \( \lambda_1^d = \{ (x = a_d) \wedge N_d \} \), we derive a new nogood \( \lambda_1 = \{ N_1 \wedge N_2 \wedge \ldots \wedge N_d \} \) where \( N_1, N_2, \ldots, N_d \) are sets of variable assignments other than \( x \)'s. [18]

In the above example, if \( V_3 \)'s domain becomes empty as a result of the nogoods \( \{(V_1 = 1), (V_2 = 2), (V_3 = 1), (V_4 = 3)\} \), \( \{(V_1 = 1), (V_2 = 2), (V_3 = 2), (V_4 = 3)\} \), and \( \{(V_1 = 1), (V_2 = 2), (V_3 = 3), (V_4 = 3)\} \), then a new nogood is derived as \( \{(V_1 = 1), (V_2 = 2), (V_4 = 3)\} \) using the nogood resolution rule. The importance of the above nogood resolution rule is two-fold. First, it informs the search that any attempt to change the assignment of variable \( x \) given the current environment would be futile as in the case for variable \( V_3 \). Second it informs the search that in order to repair this nogood, one must change the variable assignment in \( N_1 \wedge N_2 \wedge \ldots \wedge N_d \). In our example, we need to change the variable assignment of \( V_1 \) or \( V_2 \) or \( V_4 \). Indeed if we assign \( V_4 = 1 \), then it is possible to reach a solution of \( \{(V_1 = 1), (V_2 = 2), (V_3 = 3), (V_4 = 1)\} \) without difficulty. The nogood resolution also provides a theoretical support of the termination criterion for any search algorithm based upon a nogood and its resolution; for example see [13]. Informally speaking, the new nogood after resolution does not contain variable \( x \) at all and therefore includes one fewer variable. Thus we can imagine, if nogood resolution continues, the number of variables appearing
in the resolved nogoods will become fewer and fewer. As a result, whenever an empty
nogood is derived, there is no variable assignment that can be ever changed to satisfy the
problem constraints, implying that the whole problem is infeasible and that the algorithm
can terminate.

2.2 Heuristic Evaluation

Now we turn our attention to guiding the search towards constraint satisfaction. We need
to have a way of measuring the quality of each variable assignment as well as that of all the
variable assignments. We define $f_x(a)$ as the valuation function to guide the search towards
feasible solutions. In general, the valuation function is problem dependent.

**Definition 2.8.** A function $f_x(a)$ is the valuation function for a variable assignment $(x = a)$
whose value is dependent on the current assignments of the other variables in $V$. [18]

The valuation function is essential to the control mechanism in Systematic Local Search.
We follow the same definitions used in the original paper on systematic local search [18]
and classify each variable into one of the four possible classes as follows. Without loss of
generality, we place our discussions within the context of optimization by minimizing $f_x(a)$.

1. **MAXIMAL:** A variable $x$ is maximal if and only if its current assignment $(x = a)$ is
   such that $\forall b \in \Delta_x, f_x(a) \leq f_x(b)$. The current assignment for $x$ is the best possible.

2. **SUBMAXIMAL:** A variable $x$ is submaximal if and only if $\exists b \in \Delta_x$ such that $f_x(b) <
   f_x(a)$. There is a better assignment possible for $x$ from $\Delta_x$.

3. **INF:** A variable $x$ is in an infeasible state if its current assignment is $(x = a)$ but
   $a \notin \Delta_x$. The assignment is part of a known valid nogood in the presence of the other
current variable assignments.

4. **NULL:** A variable $x$ is currently not assigned a value.

We can extend the concept of a maximal assignment to maximal labels.

**Definition 2.9.** A label $\lambda = \{ (x = a) \}$ is maximal if and only if all of its variable assignments are maximal. [18]
CHAPTER 2. PRELIMINARIES

A maximal label represents a set of variable assignments in which no variable prefers a different assignment from its domain given all the other variables’ current assignments. In local search, a maximal label corresponds to a local optimum where no move which leads to improvement upon the objective function value can be found in the current solution’s neighbourhood.

The Systematic Local Search in [18] operates as follows. It keeps assigning variables to its maximal state until every variable assignment becomes maximal. This corresponds to reaching a local optimum in the context of local search. Then the search induces a nogood for every constraint violated thus forcing every variable assignment into the INF state. In local search, a nogood is induced to force the search out of a local optimum. After that it either repairs the nogood by assigning the variables to a maximal state or backtracks by unassigning some of the variables. Nogood resolution is carried out whenever there is a new nogood induced. The above process will be repeated until every variable assignment becomes maximal and no constraint is violated or an empty nogood is derived and the problem is proved infeasible.

In [18], the objective function for solving CSPs is the \textit{min-conflicts} heuristic [28], where \( f_x(a) \) is the number of constraints that will be violated when \( x \) is assigned to \( a \), given other current variable assignments. Again this reveals an intriguing link between CSPs and optimization problems as the search method tries to minimize the number of violated constraints as if it were an optimization problem. However it is not clear how a nogood should be defined in an optimization problem, since in an optimization problem there can be no constraint restricting a particular variable assignment. We need to have a clear definition for a valuation function as well as for a nogood and its resolution in optimization problems. In fact, one of the contributions of this research is the extension of all the definitions and specifications here for optimization problems, with a case study of the job shop scheduling problem [10].

Before presenting the full extension of Systematic Local Search for optimization problems, let us take another look at the important role played by the valuation function. We find it quite natural to define the valuation function as the number of violated constraints in the constraint satisfaction context. Indeed it has turned out to work quite well. So when it comes to extending the search to optimization, we can also make the objective function the valuation function as most of the local search methods typically do and rely on nogoods to diversify the search. We also make the assumption that the valuation is a single function
rather than any other form of perhaps more complex representation. Yet this assumption may oversimplify the situation. Remember that the valuation function plays a key role in controlling the search as well as providing heuristics for variable ordering. Sometimes a single function based valuation cannot provide enough powerful heuristics for choosing variable assignment under each variable class, as we in fact experienced in the case of job shop scheduling. Our objective of extending Systematic Local Search is to enrich it in such a way that not only it synthesizes desirable aspects of both constructive search and local search but it also can smoothly integrate a variety of heuristics which are found to be very effective in other stochastic search methods. For example in [4], a heuristic function to estimate a good starting point for local search is introduced along with the objective function that the local search tries to optimize. The local search alternates between searching for the optimum on the actual objective function and searching for the next best point to start another round of local search in the search space. This method with the additional valuation function is evaluated on a number of large optimization problems such as bin-packing and the results are very impressive. Therefore a more comprehensive valuation deserves further investigation and one of the interesting ways of enhancing the systematic local search is to design a better-informed valuation of the current variable assignment(s). This work of course goes beyond the scope of this thesis but we will certainly recapitulate this point in the section on future work in Chapter 6.

We will present the full Systematic Local Search for optimization problems in the next chapter.
Chapter 3

Systematic Local Search for Optimization

In this chapter, we will present a detailed description of how to extend Systematic Local Search (SysLS) for optimization problems in general.

3.1 Underlying ideas

Here we make two important assumptions about the applicability of this approach that run throughout the thesis. First, we assume that given a particular problem, there always exists a neighbourhood which keeps the complete assignment made by the search feasible. In other words, the search will concentrate on optimization without having to deal with feasibility. Second, there is only one objective to optimize. The problem should not contain multiple and perhaps conflicting objectives which are deemed to greatly complicate the search. Now let us introduce the definition for constraint satisfaction optimization problem as taken from [38]. (The constraint satisfaction optimization problem will be referred to as the optimization problem hereafter.)

Definition 3.1. A constraint satisfaction optimization problem is defined as a CSP (see Definition 2.1) plus an objective function \( f \), which maps every feasible solution to a numerical value \( (f : S \rightarrow \mathbb{R} \text{ where } S \text{ is the set of solutions to the CSP}) \).

Local search methods are generally good at producing solutions when the size of the problem becomes big but they are often trapped in local optima which can be sometimes
quite far from the optimal solutions in terms of the objective function value, especially when the landscape of the search space exhibits "big valley" phenomena [3]. When the search space contains big valley, it means that many local optima are located close to each other and are able to attract the search and prevent it from reaching the global optimum (as in [40]). It has been demonstrated that local search methods that are able to record some past information (e.g. tabu search) are generally better than those methods without memory (e.g. simulated annealing) [30]. Tabu search is equipped with a short list of recent moves which the search is forbidden to take for a short period of time. This list, known as the tabu list, is designed to prevent the search from falling back to the immediate local optimum which has just been visited. The tabu list is updated like a queue so when the capacity of the list is reached, the element that has been kept for the longest time is removed. However in the presence of big valley, the tabu list is often inadequate for escaping the local optima in the long run because the search may attempt those moves which are no longer kept in the tabu list but will lead it back to the visited area of the search space. At the same time, if we keep the tabu list too long, too many moves will become forbidden as a result and the search may not be able to move at all. Some work suggests a dynamic tabu list which varies the length of the list as the search makes progress, but it is often difficult to decide how the length of the tabu list should change during the search. The issue here is that a bounded memory cannot support both systematicity to cover the search space and freedom to move in the search space at the same time. Also we need to think of what should be recorded in memory so that the search will not waste time searching those spaces which have been visited before.

The idea is to remember those variable assignments that are essential to catch the optimization criterion when at a local optimum. For most optimization problems, a problem-specific neighbourhood in the space of complete feasible assignments is available for the search to move in. At every local optimum, we record a nogood that captures the subset of variable assignments responsible for the objective function value [10]. We represent the optimization of the objective function as a constraint, $C_{opt}$, which allows only solutions that are optimal with respect to the objective function but disallows all other solutions which are less optimal. Thanks to a neighbourhood that preserves feasibility, we can concentrate on the optimization constraint, $C_{opt}$, since it is the only constraint to be solved. Accordingly, the local evaluation function $f_x(a)$ is the value of $C_{opt}$ when $x$ is reassigned to $a$. Below we present the extended SysLS algorithm [10] for solving optimization problems.
### 3.2 The algorithm

<table>
<thead>
<tr>
<th>Input: variable set $V$ and $C_{opt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: best solution found</td>
</tr>
</tbody>
</table>

1. $\alpha \leftarrow$ initial complete assignment;
2. repeat
3. while not all $x \in \alpha$ are **MAXIMAL** do
4.   let $x = select(V)$;
5.   assign($x$);
6.   update the best solution if a better solution is found;
7. end
8. $\lambda_{\bot} = label(C_{opt})$;
9. add $\lambda_{\bot}$ to the nogood cache $\Gamma$;
10. simplify $\Gamma$ (as in [13])
11. if $\lambda_{\phi} \in \Gamma$ then return the best solution found
12. until stopping criterion;
13. return the best solution found

| Figure 3.1: SysLS for constraint optimization problems from [10] |

Here we summarize how the algorithm operates as shown in [10]. The algorithm (Figure 3.1) receives input consisting of a set of variables $V$ and the optimization constraint $C_{opt}$. It outputs the first optimal solution or a best solution found so far subject to a termination criterion customizable by an algorithm designer. The algorithm operates as follows. It starts the search with an initial complete assignment of variables $\alpha$ (line 1) which can be generated either randomly or by some problem-dependent heuristic. Then the algorithm can be decomposed into two parts. The first part basically drives the search into a local optimum (line 3-7) and the second part escapes the local optimum with the help of nogoods (line 8-11). The first part involves an inner loop structure in which the algorithm keeps attempting variable assignments until every variable becomes **MAXIMAL**. Inside this loop structure, a variable $x$ is chosen by the variable ordering procedure $select(V)$ (line 4) and then is assigned a value by the variable assignment procedure $assign(x)$ (line 5). Both of these procedures will be described in greater detail shortly. The algorithm keeps track of the best solution found so far during the search (line 6). If every variable in $\alpha$ becomes **MAXIMAL**, it means that no local variable assignment can improve the objective value and hence the search has converged to a local optimum. Then the loop condition (line 3) no longer holds and the algorithm enters the second part with a local optimum (line 8). The
second part helps the search to escape the local optimum. It consists of inducing a nogood on the local optimum for the optimization constraint $C_{opt}$ (line 8), updating the nogood cache $\Gamma$ (line 9), performing nogood resolution for all variables whose live domain is empty (line 10), and checking if the current solution is optimal by an empty nogood $\lambda_\phi$ (line 11). If an empty nogood is ever derived, no better solution can be found and the algorithm will return the optimal solution. If there is no empty nogood, then the nogood derived (from line 8) will force variable assignments in $\alpha$ from the $MAXIMAL$ state into the $INF$ state. Then the search will return to the first part. The algorithm will iterate between the first and the second part until it meets some stopping criterion; for example a pre-specified maximum number of iterations has been reached. Then the algorithm will exit and return the best solution found so far (line 13). Note that in general, the size of the nogood cache should be as big as the size of the whole search space so that SysLS is able to retain systematicity. However it is usually not practicable. Therefore we require a specification of the termination criterion as we would for all local search algorithms.

As we have mentioned, the variable ordering $select(V)$ is crucial for the overall control structure in the SysLS schema. As in Chapter 2, variables are classified into four different classes ($MAXIMAL$, $SUBMAXIMAL$, $INF$, $NULL$) according to their current assignments and live domains. We select variables based on these classes. SysLS is parameterized by the variable ordering relation similarly to the original SysLS schema [18]. Based on the four classes of variables, we need to first specify a precedence order among the classes and then an order between the variables in each class. In a local search context, a good ordering between different variable classes appears to be $NULL \prec SUBMAXIMAL \prec MAXIMAL \prec INF$, which represents a typical search cycle in local search.

First the search starts with a full assignment produced by some constructive heuristic method ($NULL \rightarrow SUBMAXIMAL$). Then it follows a local gradient to descend to a local optimum ($SUBMAXIMAL \rightarrow MAXIMAL$). After that the search must try to get out of the local optimum (triggering $MAXIMAL \rightarrow INF$ in our case).

Once a variable is chosen by the procedure $select(V)$, the action taken is specified by $assign(x)$ according to the following state transition rules for variables [18, 10].

1. If the variable is in the $MAXIMAL$ state, do nothing.

2. If the current assignment of the variable is $SUBMAXIMAL$, then assign $x$ to one of its maximal values.
Chapter 3. Systematic Local Search for Optimization

State transition in Systematic Local Search

3. If the variable is in the \textit{INF} state, i.e. the current assignment is part of a valid nogood, we switch the assignment to the maximal allowed domain element. If all elements are disallowed in the current solution, the \textit{simplify} procedure will induce a nogood based on this variable. The current assignment is unassigned and enters the \textit{NULL} state.

4. If the variable is in the \textit{NULL} state, assign it to one of its maximal values if the live domain is not empty. Otherwise no action is taken until the environment is changed.

We now describe how the state transition takes place in SysLS through a simple example illustrated in Figure 3.2. Note that in the following discussion we will not distinguish between a variable assignment and a move since both of them change a solution to another via its neighbourhood. Suppose that all the solutions in the entire search space are depicted by addition marks (+) with their associated objective function values on the vertical axis. The optimization criterion is to find a solution with the minimal objective function value (solution \(G\) marked as the global optimum in the figure). Each solution can reach one of

Figure 3.2: An illustration of state transition in Systematic Local Search
its two neighbours by moving left or right except for the leftmost and rightmost solutions. Assume that at the beginning of the search, a solution is initialized externally (i.e. the NULL state is omitted here) and the search starts at solution $S$. Clearly $S$ is in the SUBMAXIMAL state because its right-hand-side neighbour has a better objective function value. As a result, SysLS will keep moving “downhill” until it reaches the MAXIMAL state at solution $L$ where no improvement can be made in $L$’s neighbourhood. The following state transitions will be crucial for SysLS to get out of a local optimum. At solution $L$, SysLS will induce a nogood, thus forcing the search from MAXIMAL to the INF state. At the same time, variable assignment for either moving left or moving right is still allowed for $L$. Suppose that SysLS happens to choose to move right to solution $I$, $I$ itself will be in the MAXIMAL state because falling back to $L$ is disallowed by a nogood already induced and moving right to solution $M$ will deteriorate the objective function value. Again SysLS will induce a new nogood at solution $I$, thus forcing it to the INF state. At this point, the only allowed variable assignment is to move to solution $M$ since the nogood induced at $L$ forbids SysLS to move there. Once SysLS reaches $M$, it will enter the SUBMAXIMAL state again and can carry on making maximal assignment, improving the objective function value until it hits the global optimum at solution $G$.

Next we shall conduct a case study, namely the job shop scheduling problem to show how different components of SysLS can be parameterized based on a specific optimization problem. We will report details of how to realize the SysLS schema in job shop scheduling in Chapter 4 and conduct empirical evaluations in Chapter 5.
Chapter 4

Case Study: Job Shop Scheduling Problems

In this chapter, we provide a case study applying the extended Systematic Local Search to the Job Shop Scheduling Problem (JSSP).

4.1 Overview of the problem

We begin with a brief introduction of the job shop scheduling problem\(^1\). A JSSP instance has a set \( J \) of \( n \) jobs \( \{J_1, J_2, \ldots, J_n\} \) and a set \( M \) of \( m \) machines \( \{M_1, M_2, \ldots, M_m\} \). Each job is composed of a number of operations and each operation is to be processed on a machine for an uninterrupted duration. In particular, we are looking at the instances where the number of operations contained by each job is exactly the same as the total number of machines and each operation is to be processed on a different machine. The number of operations in total is \( n \times m \) since there are \( n \) jobs and each job has \( m \) operations. Other variations allow a job to have fewer than or more than \( m \) operations. An operation is denoted by \( O_{jk} \) where \( O_{jk} \) is the \( k \)th operation in the \( j \)th job \( (j \in \{1, 2, \ldots, n\}) \). The processing order between operations of the same job (denoted as \( R_J \)) is predefined and cannot be changed. To obtain a complete schedule, all operations to be processed on the same machine must be scheduled so that no two operations can execute on the same machine at the same time and the precedence relation in \( R_J \) must be observed. The objective function is used to

\(^1\)Please refer to Appendix A for a glossary of JSSP terms.
minimize the makespan whose value is the time at which the last operation finishes execution assuming that the schedule starts at time zero. It is also very convenient to represent a complete JSSP schedule as a directed acyclic graph (DAG), as shown in Figure 4.1, in which operations are represented by vertices. Precedence relations for operations of the same job and operations on the same machine are respectively represented by conjunctive and disjunctive arcs between vertices. Sometimes two vertices known as the source and the sink representing artificial operations of zero duration are placed in this DAG so that every operation has an immediate predecessor as well as an immediate successor except for the source and the sink. It is not hard to associate the length of the critical path in the DAG, which is the longest path passing from the source all the way to the sink, with the value of the makespan.

Figure 4.1 shows an instance of a JSSP which consists of 3 jobs \( (J_1, J_2 \text{ and } J_3) \) and 3 machines \( (M_1, M_2 \text{ and } M_3) \). Each job is composed of exactly 3 operations represented by vertices and each operation is processed on a different machine. Operations to be processed on the same machine are shown in the same texture. The precedence relations for operations belonging to the same job are depicted by solid lines, as for instance \( O_{11} < O_{12} < O_{13} \) for
job $J_1$. As we can also see, the operations to be processed on the same machine must be oriented such that there is no cycle in the graph as shown by the dashed lines in the figure, as for example $O_{21} \prec O_{12} \prec O_{32}$. The objective is to arrange the orientation of operations processed on the same machine for every machine such that the critical path starting from the source vertex and ending at the sink vertex is the shortest possible. The figure shows one such critical path passing through the operations (also called critical operations) shown as shaded vertices ($O_{11} \rightarrow O_{22} \rightarrow O_{23}$).

The Job Shop Scheduling Problem is NP-hard. Since its first introduction in the 1960s, a great number of methods that stem from a wide range of different backgrounds, including Operations Research [27] and Machine Learning [43], have been proposed to solve the problem. Here we will focus our attention on heuristic search approaches.

In [26], a standard simulated annealing method is proposed to solve the JSSP. Simulated annealing has an embedded local search procedure which keeps moving to schedules with better makespan values but also accepts schedules with worse makespan values in a controlled manner as a way of getting out of local optima. Simulated annealing has been applied successfully to other combinatorial optimization problems, but a general difficulty with simulated annealing is that it has parameters such as initial temperature and cooling rate and there are only limited general guidelines on how to tune them. More often than not these parameters are tuned by trial-and-error, resulting in less than optimal parameter settings and the results may vary significantly given different settings. When it comes to a specific problem such as a JSSP, we must tune the parameters well in order to obtain good performance. Take the cooling rate for example: if we cool the algorithm too fast, it will converge to a local optimum prematurely. On the other hand, if the cooling rate is too slow, the algorithm may take too long to converge. Finding a good balance between performance and running-time is not trivial and has led to an entirely different research topic for heuristic search [20]. Also in [26] it is proved that if we restrict our neighbourhood to an exchange of two adjacent critical operations processed on the same machine only, then we can start from an arbitrary feasible schedule and get to an optimal schedule by moving through this neighbourhood. More recently, simulated annealing has been hybridized with genetic algorithms in [24] to achieve better performance for JSSPs.

Genetic algorithms have also been applied to the JSSP in [8]. Inspired by the principle of the “survival of the fittest” from natural evolution, a genetic algorithm works on a set of solutions known as a “population” and uses “crossover” and “mutation” to select an
individual solution according to its objective function value and to evolve the whole population over time. In principle good “genes” corresponding to components in a good solution survive while less optimal solutions become extinct and the whole population converges to the optimal eventually. The advantage of the genetic algorithm is that it is able to employ a concept of population to explore a vast area of the search space efficiently although at a cost of slow convergence. It can achieve very good performance when the problem has very few constraints. However, the problem with the genetic algorithm in the case of the JSSP is that crossover and mutation may produce infeasible solutions, i.e. schedules with cycles. The remedy is to either carefully restrict the neighbourhood by avoiding infeasible solutions or fix any infeasible solutions after crossover or mutation. However neither remedy is ideal since it either compromises the algorithm’s performance or it is very expensive to run. Another problem associated with the genetic algorithm in general is that the “evolution rate” of the population must be carefully engineered. Usually at a given point in the search in a genetic algorithm, the population can have multiple copies of several good individuals that may turn out to be actually suboptimal. If the evolution is not well designed, these individuals may prematurely dominate the population, and it is very difficult to further evolve the population when it is occupied by only a few individuals resulting from poor diversification in the search. Incidentally it is not a common practice to do a restart in a genetic algorithm since a genetic algorithm is already quite expensive to run.

The most prevalent and successful approaches so far come from tabu search [12]. Tabu search has a short-term memory (also known as a tabu list) to record and forbid a list of recent moves so as to help the search escape local optima. A rule of thumb is to keep the length of the tabu list short in order to achieve a good balance between escaping from the local optima and exploiting the search space. Since tabu search has only one major parameter (the length of the tabu list), it is relatively easy to tune and usually the results turn out to be insensitive to the variance of the tabu list length. In order to obtain high-quality results for the JSSP, tabu search was specifically hybridized with other techniques [9, 11, 29, 31], in particular the shifting bottleneck procedure [1]. The shifting bottleneck procedure works as follows. It sequences all the machines one by one by identifying a machine as a bottleneck at each step. Once operations on the bottleneck machine are scheduled, the operations on all previously scheduled machines will be re-optimized locally using methods for solving certain one-machine scheduling problems. Since this procedure is a little computationally expensive, it can be embedded in a tabu search and called when
the search gets stuck in local optima as in [31]. Another popular technique is to restart
the tabu search from several good solutions kept during the search (also known as elite
solutions) as in [29], since it is perceived that these elite solutions are generally located in
promising regions of the search space when compared to randomly generated solutions. So
as soon as a single run of the tabu search cannot make any further progress on solution
quality after a certain number of iterations, another run of tabu search starting from one
of the elite solutions will be initiated. Since the number of elite solutions is fixed and we
only update them when the objective function value gets better, the search will eventually
run out of elite solutions and terminate. Other techniques include varying the length of the
tabu list during search [9, 31] and parallel execution of the tabu search [11]. For a more
comprehensive survey of the job shop scheduling problem and its various solving methods,
please refer to [21].

4.2 Systematic Local Search

We will describe the approach of applying SysLS to the JSSP in two steps. First we shall
model the JSSP as a constraint optimization problem according to Definition 3.1 in terms of
variables, domain, constraints and optimization criterion. The modelling will be presented
in the next two subsections. Then we will discuss how to realize different components of
SysLS (Figure 3.1) in solving a JSSP for efficiency and performance.

4.2.1 Variables and domains

Here we formulate the JSSP as a constraint optimization problem by first defining variables
and their domains. In order to produce a complete schedule, the processing order of oper-
ations on each machine must be assigned. A variable \( v \) is defined as the ordering between
two different operations processed on the same machine. For every variable, the domain \( D \)
contains only two values \( D = \{<, >\} \) specifying the ordering.

Since each of the \( m \) machines has \( n \) operations, the total number of variables is \( \frac{mn(n-1)}{2} \)
according to the above definition. We denote this variable set as \( V^0 \). Note that it is not
necessary to define variables with respect to the ordering between two operations processed
on different machines as this is enforced by the given precedence constraints of operations
belonging to the same job \( R_j \). Although all the domains contain just two values, the size of
the variable set is quadratic with the number of operations on each machine. We will show
how to manage this size and keep it reasonably small. We should point out that there exists
an alternative way of modelling the scheduling problem as a CSP in which the variable is
defined as the start time of each operation and the corresponding domain contains possible
values of time at which the operation can start executing. This way of modelling makes the
number of variables exactly the same as the total number of operations. However following
the advice in [37], we chose to impose precedence constraints between operations as they do
not explicitly specify the start time of each operation until an actual schedule is constructed.
This offers much flexibility during the search. Also, as we will see shortly, the precedence
constraints make it easy for us to define a relatively simple and short nogood for SysLS.

A closer look reveals that the variable set \( V^0 \) is redundant as some of its variable assign-
ments can be deduced from other variable assignments using the transitive property of the
precedence relation\(^2\). Since the processing order imposed by a permutation of \( n \) operations
on each machine is enough to determine a complete schedule, we only need \( m \times (n - 1) \)
variables and their precedence assignments to construct a complete schedule. We denote
such a variable set as \( V^1 \). It only contains those variables defined as the ordering between
every two adjacent operations processed on the same machine. It is obvious that \( V^1 \subseteq V^0 \)
due to the transitive property of the precedence relation.

Once the variables of \( V^1 \) are assigned and every operation is processed at its earliest start
time, a complete schedule is constructed. In such a schedule, a critical path is constructed
by following the precedence relations specified by both \( V^1 \) and \( R_J \). In [26], it was shown
that only changing the assignment of variables on the critical paths can guarantee to keep
the schedule feasible and can contribute to reducing the makespan value. Now we further
restrict the set of variables from \( V^1 \) to those on the critical path and denote these variables
as \( V^2 \) and clearly \( V^2 \subseteq V^1 \subseteq V^0 \). It is further shown in [29] that if the critical path is
partitioned into blocks of operations processed on the same machine, then only changing
the assignment of variables on the borderline of a block can result in immediate makespan
improvement. A variable is on the borderline of a block if it is the last variable of the
first block, the first variable of the last block or the first or the last variable of any block
in the middle provided that the block contains at least one variable. Such a variable set
is denoted as \( V^3 \) and \( V^3 \subseteq V^2 \subseteq V^1 \subseteq V^0 \). For variable set \( V^3 \) and \( V^2 \), the number of
variables contained are usually far fewer than \( m \times (n - 1) \) thus alleviating the problem of

\(^2\text{If } a < b \text{ and } b < c \text{ then } a < c \text{ and similar for } "\text{>}".\)
large variable sets in the modelling.

4.2.2 Constraints and the Optimization Criterion

In a JSSP, the constraint says that no two operations can overlap during their execution on a machine at any given time. Also a feasible schedule should have no cycles in the corresponding DAG. If we only change variable assignment in $V^2$ or $V^3$, then it is guaranteed that every schedule generated is kept feasible [26, 29]. Relying on such a feasible neighbourhood, we can focus our attention on improving upon the objective, i.e., minimizing the makespan without having to worry about satisfying constraints. Therefore the optimization criterion ($C_{opt}$) is intended to minimize the makespan of the schedule. The SysLS algorithm takes the heuristic valuation function $f_x(a)$ of a variable assignment ($\langle x = < \rangle$ or $\langle x = > \rangle$) to be the makespan of the schedule produced by the assignment of the variable, with other current variable assignments being fixed. At a local optimum, SysLS induces a nogood on the subset of variable assignments responsible for the makespan. Since the makespan is solely determined by the length of the critical path in the schedule, the nogood contains only variable assignments representing the precedence assignments along the critical path, previously referred to as variable set $V^2$.

Next we shall prove that nogood resolution on such nogoods is sound, meaning that induced nogoods do not preclude the solution containing the minimal makespan. Note that both the theorem and the proof come from [10]3.

**Theorem 4.1.** In a JSSP, if any nogood recorded at a local optimum represents the variable assignment on the critical path (i.e. the variable assignment of $V^2$), then this nogood resolution is sound with respect to finding the schedule with the optimal makespan.

**Proof.** First recall that the assignments (i.e. the processing order of two different operations processed on the same machine) of variables in $V^2$ represent a critical path in a network of precedence relations drawn from $V^1$ and the precedence relations specified by the jobs $R_j$. A nogood is induced if all the variable assignments in $V^2$ become maximal which means reversing any of the precedence relations will increase the makespan. Suppose that the nogood induced is $\lambda_1 = \{ x = a \}_{x \in V^2}$ where $a \in \{<, >\}$. At this stage, any variable in the nogood can be chosen and assigned another value from its live

---

3The proof contributed to, but was not published in, [10]. The proof will appear in a paper currently being prepared for publication in a journal.
domain $\Delta$, provided that $\Delta \neq \emptyset$. Without loss of generality, let $x_0$ be the chosen variable with assignment $(x_0 = \leftarrow)$ and denote all other variable assignments in $\lambda_1^1$ as $N^1$. Then $\lambda_1^1 = \{(x_0 = \leftarrow) \land N^1\}$. Meanwhile, we associate the length of the critical path with the nogood and record it as $L(\lambda_1^1)$. As long as the label of the nogood is valid, the path represented by the variable assignment in $\lambda_1^1$ will always be present in the DAG, corresponding to the current solution. Therefore the length of the critical path $L$ in the precedence network satisfies $L \geq L(\lambda_1^1)$ as long as all assignments from $\lambda_1^1$ are current irrespective of other variable assignments.

If another nogood $\lambda_1^2$ is induced such that $\lambda_1^2 = \{(x_0 = \rightarrow) \land N^2\}$ with critical path length $L(\lambda_1^2)$, then by resolution a new nogood $\lambda_1 = \{N^1 \land N^2\}$ can be induced with the critical path length $L(\lambda_1) \geq \min(L(\lambda_1^1), L(\lambda_1^2))$. The proof for $L(\lambda_1)$ is fairly straightforward. Suppose that all variable assignments in nogood $\lambda_1$ are valid, the assignment of variable $x_0$ must be either $(x_0 = \leftarrow)$ or $(x_0 = \rightarrow)$. In the former case, the length of the critical path $L_{(x_0 = \leftarrow) \land N^1 \land N^2} \geq L_{(x_0 = \leftarrow) \land N^1} = L(\lambda_1^1)$. In the latter case, $L_{(x_0 = \rightarrow) \land N^1 \land N^2} \geq L_{(x_0 = \rightarrow) \land N} = L(\lambda_1^2)$. Therefore, if $\lambda_1$ is valid, the length of any critical path $L_{N^1 \land N^2} \geq \min(L(\lambda_1^1), L(\lambda_1^2))$. In other words, $\lambda_1$ prevents solutions only with same or worse makespan than the ones that have already been found. By induction, if ever an empty nogood $\lambda_\emptyset$ is derived, then the associated critical path length $L(\lambda_\emptyset)$ is the minimal length of the critical path in any solution and hence is the optimal makespan since no shorter critical path can be ever obtained by changing any variable assignment. Therefore, resolution of nogoods representing critical paths is sound with respect to optimizing the makespan.

Next we describe how to instantiate the different components of the algorithm in order to solve a JSSP efficiently.

4.2.3 Algorithm Design

To apply SysLS to a JSSP, we need to implement the four parameterizable components of the algorithm shown in Figure 3.1. Now we describe them in turn.

1. initial solution - We use a very simple method to produce the initial solution. Observe that an operation $O_{jk}$ has its job ordering fixed by $k$ due to $R_j$. Within the same job, an operation with a smaller value of $k$ must be executed before an operation with a larger value of $k$ though they are to be scheduled on different machines. It is quite intuitive that an operation with a smaller value of $k$ should be given the preference
of being processed earlier in order to make the processing of operations with a larger value of \( k \) take place as soon as possible. Therefore we impose the following ordering among operations processed by the same machine. For each machine \( m \in M \) and two operations \( O_{j_1 k_1} \) and \( O_{j_2 k_2} \) (\( j_1 \neq j_2 \)):

\[
\begin{align*}
O_{j_1 k_1} &< O_{j_2 k_2} & \text{ if } k_1 < k_2; \\
O_{j_1 k_1} &> O_{j_2 k_2} & \text{ if } k_1 > k_2; \\
\text{choose } O_{j_1 k_1} &< O_{j_2 k_2} \text{ or } O_{j_1 k_1} > O_{j_2 k_2} & \text{ arbitrarily if } k_1 = k_2.
\end{align*}
\]

Once the orderings are obtained on every machine, a complete schedule can be constructed based on the earliest start time of each operation.

2. **select**(\( V \)) - For optimization problems, we prefer using the precedence ordering \( \text{NULL} < \text{SUBMAXIMAL} < \text{INF} \). Note that \( \text{MAXIMAL} \) is not included here because there is no action to take for \( \text{MAXIMAL} \) variables. As we have discussed, this ordering represents a typical cycle in local optimization. The search continuously accepts an improving move, gets trapped in a local optimum, and then is forced out by an induced nogood. The variable ordering within the variable class \( \text{SUBMAXIMAL} \) is best-improvement. This can be summed up as choosing a variable \( x \) whose reassignment leads to the most significant improvement of the makespan value with respect to the current assignment of \( x \). Choosing among \( \text{INF} \) variables in order to escape the local optimum is less straightforward. Under class \( \text{INF} \), we implement a recency-based diversification scheme similar to that used frequently in tabu search [15]. We associate each variable assignment with a time-stamp which records the latest time (implemented as the iteration count) that the variable assignment is present in the solution. Initially every variable assignment’s time-stamp is set to zero. Preference among the \( \text{INF} \) variables is first given to those variables whose assignment has never been changed since the start of the search. Ties are broken by the better makespan value. Once a variable assignment is selected, the time-stamp is updated with the iteration count. If all the variable assignments of \( V^2 \) have been encountered before (i.e. every variable assignment has a time-stamp greater than zero), then the preference is given to the assignment with the oldest time stamp. We never choose an \( \text{INF} \) variable whose live domain is empty. It is rare that the search will enter the \( \text{NULL} \) state in which some of the variables’ live domains become empty. In such a case, the idea is to assign values for the null variables as soon as possible.
3. \textbf{assign}(x) - In Chapter 3, we pointed out that the action taken by the function \textit{assign}(x) depends on the state of the variable. This function will be called only for variables whose live domain is not empty. Note that in a JSSP, choosing a value to assign is greatly simplified in that the domain contains just two values. Once a variable is chosen for reassignment, we simply assign the other value to the variable.

4. \textbf{stopping criterion} - We can add extra flexibility into the SysLS algorithm for a JSSP by introducing an additional stop condition. The standard termination condition is to execute a pre-specified maximum number of iterations. For a JSSP, it has been shown in [29] that if all the operations on the critical path are processed on one machine or belong to one job, then the makespan is optimal. This also means that the optimal value hits the lower bound. We add this into the stop conditions as well.

We will present experimental results on benchmark JSSPs as well as study the SysLS's run-time performance in the next chapter.
Chapter 5

Experimental Results

5.1 Benchmark instances

The empirical evaluation of SysLS was conducted on a total of 58 well-known benchmark JSSP instances which have also been used extensively by others, as for example in [26, 9, 11, 8, 29, 24, 31, 12, 16]. Each instance is characterized by a number of jobs ($n$) and a number of machines ($m$) with the duration for each of the $n \times m$ operations randomly set between 0 and 100. These benchmark instances are ft06, ft10, ft20, orb01-10, abz5-9 and la01-40. We compared the best makespan value ($L_{min}$) obtained by SysLS with the optimal makespan value or its lower bound $L_{opt}$ and reported the relative error, $RE\% = \frac{(L_{min}-L_{opt})}{L_{opt}} \times 100$, in Tables 5.1 and 5.2. If the optimal makespan value for an instance was found by the SysLS, the value is marked by an asterisk (*). We also reported the mean makespan value obtained over 10 runs, the mean relative error and the mean run-time. The maximum number of iterations for stopping the algorithm was set to 20000. All the experiments were run on a Sun Blade 2000 workstation with a 1.2 GHz UltraSPARC III processor and 2 GB memory.

5.2 Experimental Results

Tables 5.1 shows that SysLS performed very well on the 40 Lawrence instances (la01-40), 28 of which were solved optimally and that the best makespan values for the rest of them

---

1 All of the benchmarks are available at http://people.brunel.ac.uk/~mastijb/jeb/info.html.
Table 5.1: Experimental results of SysLS on problem instances ln01-40

<table>
<thead>
<tr>
<th>$n \times m$</th>
<th>Instance</th>
<th>$L_{opt}$/(LB,UB)</th>
<th>$L_{min}$</th>
<th>$RE_{min}$%</th>
<th>$L_{avg}$</th>
<th>$RE_{avg}$%</th>
<th>$t_{avg}$(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 \times 5</td>
<td>la01</td>
<td>666</td>
<td>666*</td>
<td>0</td>
<td>666</td>
<td>0</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>la02</td>
<td>655</td>
<td>655*</td>
<td>0</td>
<td>655</td>
<td>0</td>
<td>96.32</td>
</tr>
<tr>
<td></td>
<td>la03</td>
<td>597</td>
<td>597*</td>
<td>0</td>
<td>597</td>
<td>0</td>
<td>97.54</td>
</tr>
<tr>
<td></td>
<td>la04</td>
<td>590</td>
<td>590*</td>
<td>0</td>
<td>590</td>
<td>0</td>
<td>115.73</td>
</tr>
<tr>
<td></td>
<td>la05</td>
<td>593</td>
<td>593*</td>
<td>0</td>
<td>593</td>
<td>0</td>
<td>0.05</td>
</tr>
<tr>
<td>15 \times 5</td>
<td>la06</td>
<td>926</td>
<td>926*</td>
<td>0</td>
<td>926</td>
<td>0</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>la07</td>
<td>890</td>
<td>890*</td>
<td>0</td>
<td>890</td>
<td>0</td>
<td>230.70</td>
</tr>
<tr>
<td></td>
<td>la08</td>
<td>863</td>
<td>863*</td>
<td>0</td>
<td>863</td>
<td>0</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>la09</td>
<td>951</td>
<td>951*</td>
<td>0</td>
<td>951</td>
<td>0</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>la10</td>
<td>958</td>
<td>958*</td>
<td>0</td>
<td>958</td>
<td>0</td>
<td>0.05</td>
</tr>
<tr>
<td>20 \times 5</td>
<td>la11</td>
<td>1222</td>
<td>1222*</td>
<td>0</td>
<td>1222</td>
<td>0</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>la12</td>
<td>1039</td>
<td>1039*</td>
<td>0</td>
<td>1039</td>
<td>0</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>la13</td>
<td>1150</td>
<td>1150*</td>
<td>0</td>
<td>1150</td>
<td>0</td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td>la14</td>
<td>1292</td>
<td>1292*</td>
<td>0</td>
<td>1292</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>la15</td>
<td>1207</td>
<td>1207*</td>
<td>0</td>
<td>1207</td>
<td>0</td>
<td>23.95</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>la16</td>
<td>945</td>
<td>945*</td>
<td>0</td>
<td>945.8</td>
<td>0.085</td>
<td>222.04</td>
</tr>
<tr>
<td></td>
<td>la17</td>
<td>784</td>
<td>784*</td>
<td>0</td>
<td>784</td>
<td>0</td>
<td>188.74</td>
</tr>
<tr>
<td></td>
<td>la18</td>
<td>848</td>
<td>848*</td>
<td>0</td>
<td>848</td>
<td>0</td>
<td>223.42</td>
</tr>
<tr>
<td></td>
<td>la19</td>
<td>842</td>
<td>842*</td>
<td>0</td>
<td>844.7</td>
<td>0.32</td>
<td>225.38</td>
</tr>
<tr>
<td></td>
<td>la20</td>
<td>902</td>
<td>907</td>
<td>0.55</td>
<td>909.8</td>
<td>0.86</td>
<td>253.87</td>
</tr>
<tr>
<td>15 \times 10</td>
<td>la21</td>
<td>1046</td>
<td>1055</td>
<td>0.86</td>
<td>1062.3</td>
<td>1.56</td>
<td>586.72</td>
</tr>
<tr>
<td></td>
<td>la22</td>
<td>927</td>
<td>935</td>
<td>0.86</td>
<td>942.9</td>
<td>1.72</td>
<td>571.16</td>
</tr>
<tr>
<td></td>
<td>la23</td>
<td>1032</td>
<td>1032*</td>
<td>0</td>
<td>1032</td>
<td>0</td>
<td>62.24</td>
</tr>
<tr>
<td></td>
<td>la24</td>
<td>935</td>
<td>944</td>
<td>0.96</td>
<td>954.8</td>
<td>2.12</td>
<td>468.14</td>
</tr>
<tr>
<td></td>
<td>la25</td>
<td>977</td>
<td>979</td>
<td>0.20</td>
<td>985.5</td>
<td>0.87</td>
<td>553.90</td>
</tr>
<tr>
<td>20 \times 10</td>
<td>la26</td>
<td>1218</td>
<td>1218*</td>
<td>0</td>
<td>1218</td>
<td>0</td>
<td>860.63</td>
</tr>
<tr>
<td></td>
<td>la27</td>
<td>1235</td>
<td>1255</td>
<td>1.62</td>
<td>1257.6</td>
<td>1.83</td>
<td>1299.88</td>
</tr>
<tr>
<td></td>
<td>la28</td>
<td>1216</td>
<td>1216*</td>
<td>0</td>
<td>1216</td>
<td>0</td>
<td>1236.23</td>
</tr>
<tr>
<td></td>
<td>la29</td>
<td>(1142,1153)</td>
<td>1177</td>
<td>3.06</td>
<td>1194.3</td>
<td>4.58</td>
<td>1320.14</td>
</tr>
<tr>
<td></td>
<td>la30</td>
<td>1355</td>
<td>1355*</td>
<td>0</td>
<td>1355</td>
<td>0</td>
<td>461.51</td>
</tr>
<tr>
<td>30 \times 10</td>
<td>la31</td>
<td>1784</td>
<td>1784*</td>
<td>0</td>
<td>1784</td>
<td>0</td>
<td>99.72</td>
</tr>
<tr>
<td></td>
<td>la32</td>
<td>1850</td>
<td>1850*</td>
<td>0</td>
<td>1850</td>
<td>0</td>
<td>72.46</td>
</tr>
<tr>
<td></td>
<td>la33</td>
<td>1719</td>
<td>1719*</td>
<td>0</td>
<td>1719</td>
<td>0</td>
<td>68.00</td>
</tr>
<tr>
<td></td>
<td>la34</td>
<td>1721</td>
<td>1721*</td>
<td>0</td>
<td>1721</td>
<td>0</td>
<td>301.51</td>
</tr>
<tr>
<td></td>
<td>la35</td>
<td>1888</td>
<td>1888*</td>
<td>0</td>
<td>1888</td>
<td>0</td>
<td>310.01</td>
</tr>
<tr>
<td>15 \times 15</td>
<td>la36</td>
<td>1268</td>
<td>1275</td>
<td>0.55</td>
<td>1291.3</td>
<td>1.84</td>
<td>1010.31</td>
</tr>
<tr>
<td></td>
<td>la37</td>
<td>1397</td>
<td>1411</td>
<td>1.00</td>
<td>1430.6</td>
<td>2.41</td>
<td>1139.81</td>
</tr>
<tr>
<td></td>
<td>la38</td>
<td>1196</td>
<td>1208</td>
<td>1.00</td>
<td>1239.0</td>
<td>3.60</td>
<td>1056.95</td>
</tr>
<tr>
<td></td>
<td>la39</td>
<td>1233</td>
<td>1250</td>
<td>1.38</td>
<td>1255.9</td>
<td>1.86</td>
<td>1005.03</td>
</tr>
<tr>
<td></td>
<td>la40</td>
<td>1222</td>
<td>1239</td>
<td>1.39</td>
<td>1250.6</td>
<td>2.34</td>
<td>1079.48</td>
</tr>
</tbody>
</table>
CHAPTER 5. EXPERIMENTAL RESULTS

Table 5.2: Experimental results of SysLS for problem instances ft06, ft10, ft20, orb01–10 and abz5–9 (18 instances in total)

<table>
<thead>
<tr>
<th>Instance (n x m)</th>
<th>$L_{opt}$ or (LB, UB)</th>
<th>$L_{min}$</th>
<th>RE$_{min}$%</th>
<th>$L_{avg}$</th>
<th>RE$_{avg}$%</th>
<th>$t_{avg}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft06 (6 x 6)</td>
<td>55</td>
<td>55*</td>
<td>0</td>
<td>55</td>
<td>0</td>
<td>4.55</td>
</tr>
<tr>
<td>ft10 (10 x 10)</td>
<td>930</td>
<td>938</td>
<td>0.86</td>
<td>956.4</td>
<td>2.84</td>
<td>234.77</td>
</tr>
<tr>
<td>ft20 (20 x 5)</td>
<td>1165</td>
<td>1165*</td>
<td>0</td>
<td>1179.5</td>
<td>1.24</td>
<td>563.27</td>
</tr>
<tr>
<td>orb01 (10 x 10)</td>
<td>1059</td>
<td>1060</td>
<td>0.10</td>
<td>1086.3</td>
<td>2.58</td>
<td>247.99</td>
</tr>
<tr>
<td>orb02 (10 x 10)</td>
<td>888</td>
<td>889</td>
<td>0.11</td>
<td>891.5</td>
<td>0.39</td>
<td>226.06</td>
</tr>
<tr>
<td>orb03 (10 x 10)</td>
<td>1005</td>
<td>1022</td>
<td>1.69</td>
<td>1041.4</td>
<td>3.62</td>
<td>251.85</td>
</tr>
<tr>
<td>orb04 (10 x 10)</td>
<td>1005</td>
<td>1011</td>
<td>0.60</td>
<td>1024.5</td>
<td>1.94</td>
<td>243.39</td>
</tr>
<tr>
<td>orb05 (10 x 10)</td>
<td>887</td>
<td>887*</td>
<td>0</td>
<td>894.0</td>
<td>0.79</td>
<td>215.74</td>
</tr>
<tr>
<td>orb06 (10 x 10)</td>
<td>1010</td>
<td>1013</td>
<td>0.30</td>
<td>1022.1</td>
<td>1.20</td>
<td>247.96</td>
</tr>
<tr>
<td>orb07 (10 x 10)</td>
<td>397</td>
<td>397*</td>
<td>0</td>
<td>403.3</td>
<td>1.59</td>
<td>199.00</td>
</tr>
<tr>
<td>orb08 (10 x 10)</td>
<td>899</td>
<td>899*</td>
<td>0</td>
<td>928.0</td>
<td>3.23</td>
<td>251.99</td>
</tr>
<tr>
<td>orb09 (10 x 10)</td>
<td>934</td>
<td>934*</td>
<td>0</td>
<td>943.9</td>
<td>1.06</td>
<td>236.27</td>
</tr>
<tr>
<td>orb10 (10 x 10)</td>
<td>944</td>
<td>944*</td>
<td>0</td>
<td>950.6</td>
<td>0.70</td>
<td>201.33</td>
</tr>
<tr>
<td>abz5 (10 x 10)</td>
<td>1234</td>
<td>1236</td>
<td>0.16</td>
<td>1243.2</td>
<td>0.75</td>
<td>233.41</td>
</tr>
<tr>
<td>abz6 (10 x 10)</td>
<td>943</td>
<td>943*</td>
<td>0</td>
<td>954.3</td>
<td>1.20</td>
<td>244.64</td>
</tr>
<tr>
<td>abz7 (20 x 15)</td>
<td>656</td>
<td>680</td>
<td>3.66</td>
<td>690.5</td>
<td>5.26</td>
<td>2561.76</td>
</tr>
<tr>
<td>abz8 (20 x 15)</td>
<td>(645, 669)</td>
<td>683</td>
<td>5.89</td>
<td>705.5</td>
<td>9.38</td>
<td>2647.40</td>
</tr>
<tr>
<td>abz9 (20 x 15)</td>
<td>(661, 679)</td>
<td>704</td>
<td>6.51</td>
<td>715.4</td>
<td>8.23</td>
<td>2373.62</td>
</tr>
</tbody>
</table>

were generally within 2% of the optimal or its lower bound. For the three ft instances (Table 5.2), we were able to solve two of them to optimality while the best result for the notoriously difficult ft10 was only 1% from optimal. For instances orb (Table 5.2), SysLS was able to find very good solutions: 5 out of 10 instances were solved optimally while the other 4 instances were within 1% of the optimal and only one was off the 1% range (1.69%). For the five abz instances (Table 5.2), one achieved an optimal makespan (abz6), another one (abz5) was nearly optimal (best within 0.5%) while instances abz7–9 remained challenging.

Next we compared our SysLS results with the best results reported by some standard heuristic approaches as well as some very specialized ones. First we compared our results with standard stochastic local search algorithms including simulated annealing (SA) [26] and tabu search (TB) [12]. We also selected three state-of-the-art specialized approaches for comparison. The first (TSAB) [29] is a fast tabu search algorithm with the neighbourhood restricted to $V^3$ only. It also intensifies the search by restarting from elite solutions. The second tabu search method (TSSB) [31] uses the shifting bottleneck procedure which is a
very effective heuristic for a JSSP in terms of generating initial solutions as well as refining subsequent ones. It also dynamically modifies the length of the tabu list during the search. The third method (SAGen) [24] actually combines a genetic algorithm with simulated annealing, which allows reheating after cooling in the simulated annealing phase. However, its experimental results are only available for a few hard JSSP instances.

Table 5.3 shows the comparison of the best makespan values found by all the algorithms considered, including SysLS across the 40 Lawrence benchmark instances. The best makespan values are grouped according to different problem sizes. The comparison clearly shows that for these benchmark problems SysLS consistently found better solutions than the general local search methods, namely simulated annealing and tabu search. SysLS was able to solve 28 out of 40 instances to optimality, while simulated annealing solved 23 instances optimally and tabu search scored only 16. In terms of the mean relative error, the gap between best found and the optimal is also smaller: overall 0.35 for SysLS versus 0.74 for simulated annealing and 1.77 for tabu search respectively. When comparing the mean relative error for each instance group of the same problem size, SysLS is consistently better than simulated annealing and tabu search. This result is encouraging and demonstrates that SysLS is a very competitive general search method. Nevertheless SysLS has yet to close its performance gap with the very specialized heuristic methods for the JSSP described in [29, 31].
CHAPTER 5. EXPERIMENTAL RESULTS

Table 5.4: A comparison of SysLS with other heuristic approaches on the best makespan found for a total of 11 very hard JSSP instances. The notation “—” means no result is available for the selected instance. The MRE is the mean relative error over the selected 11 instances.

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>SysLS</th>
<th>SA</th>
<th>TB</th>
<th>TSAB</th>
<th>TSSB</th>
<th>SAGen</th>
</tr>
</thead>
<tbody>
<tr>
<td>la19</td>
<td>842</td>
<td>848</td>
<td>860</td>
<td>842</td>
<td>842</td>
<td>842</td>
</tr>
<tr>
<td>la21</td>
<td>1055</td>
<td>1063</td>
<td>1099</td>
<td>1047</td>
<td>1046</td>
<td>1047</td>
</tr>
<tr>
<td>la24</td>
<td>944</td>
<td>952</td>
<td>989</td>
<td>939</td>
<td>938</td>
<td>938</td>
</tr>
<tr>
<td>la25</td>
<td>979</td>
<td>992</td>
<td>995</td>
<td>977</td>
<td>979</td>
<td>977</td>
</tr>
<tr>
<td>la27</td>
<td>1255</td>
<td>1269</td>
<td>1258</td>
<td>1236</td>
<td>1235</td>
<td>1236</td>
</tr>
<tr>
<td>la29</td>
<td>1177</td>
<td>1218</td>
<td>1206</td>
<td>1160</td>
<td>1168</td>
<td>1167</td>
</tr>
<tr>
<td>la36</td>
<td>1275</td>
<td>1293</td>
<td>1302</td>
<td>1268</td>
<td>1268</td>
<td>1268</td>
</tr>
<tr>
<td>la37</td>
<td>1411</td>
<td>1433</td>
<td>1453</td>
<td>1407</td>
<td>1411</td>
<td>1401</td>
</tr>
<tr>
<td>la38</td>
<td>1208</td>
<td>1215</td>
<td>1254</td>
<td>1196</td>
<td>1201</td>
<td>1201</td>
</tr>
<tr>
<td>la39</td>
<td>1250</td>
<td>1248</td>
<td>1269</td>
<td>1233</td>
<td>1240</td>
<td></td>
</tr>
<tr>
<td>la40</td>
<td>1239</td>
<td>1234</td>
<td>1261</td>
<td>1229</td>
<td>1233</td>
<td>1226</td>
</tr>
</tbody>
</table>

| MRE              | 1.09  | 2.13| 3.63| 0.23 | 0.52 |      |

Table 5.4 compares the performance of all the selected algorithms on a number of very hard instances. Once again it shows that SysLS outperforms standard simulated annealing and tabu search on most of the instances. More importantly it reveals that SysLS also compares favourably with specialized heuristic methods tailored for JSSPs. For example, for instance la19, SysLS achieved the same result (the optimal) as all three specialized algorithms did. Also SysLS was able to obtain the same near-optimal results as tabu search with the shifting bottleneck procedure (TSSB) on instance la25 and la37. Unlike other approaches which are designed specially for the JSSP, the SysLS algorithm remains relatively general.

5.3 Runtime Performance

So far we have seen SysLS can produce good results at the end of the search process. Now let us turn our attention to what is happening during the course of the search, which is more important in terms of further improvement of the algorithm’s performance. In order to study the SysLS’s runtime performance, we record the improvement on the best makespan value found so far at every 10 iterations and let SysLS run on a particular instance for a maximum of 2000 iterations. The objective here is no longer to find the best makespan value
Figure 5.1: Box plot for the aggregated run-time performance on la21 over 10 runs. The maximum number of iterations is set to 2000.
Figure 5.2: Box plot for the aggregated run-time performance on orb04 over 10 runs. The maximum number of iterations is set to 2000.
possible but to show how SysLS makes progress over time by observing the improvement on the best makespan value found so far. Each instance is run 10 times and we use a box plot\textsuperscript{3} to show the lower quartile, the median, and the upper quartile values for the best makespan found at every 10 iterations. Results are shown in Figure 5.1 and Figure 5.2 for benchmark instances la21 and orb04 respectively. Similar results were observed for all other benchmark instances.

The steep descent of the curve in both figures shows that SysLS makes rapid improvement in the makespan during approximately the first 200 iterations. SysLS is subsequently able to make continual improvement over time but at a much slower rate. This reveals two things about the heuristics guiding the search in SysLS. First, given more time (or more iterations), SysLS will continue to improve the makespan value as suggested by the trend of the curve. On the other hand, unlike what is happening during the first few iterations, SysLS’s progress on the makespan value becomes much slower as the number of iterations increases, thus blurring the line between search progress and search stagnation where the search almost stops improving the objective value and it becomes more desirable to restart the search than to carry it on. In Chapter 6 we will propose two methods for improving SysLS’s runtime performance and preventing search stagnation as future research directions.

The main feature that separates SysLS from other heuristic methods is the use of nogoods to escape local optima. In order to gain a deeper insight into how effectively the nogood can diversify the search, we plotted all the nogoods ever induced by SysLS in relation to the progress made on the makespan value for a single run of the SysLS. Figure 5.3 and Figure 5.4 are based on experiments conducted on instance la01 with 10 jobs by 5 machines. We chose a rather simple instance here because we wanted to demonstrate clearly a general trend that has been observed on many other instances.

Now let us turn our attention to Figure 5.3 and describe how it was generated. First all the nogoods induced during the search were sorted in ascending order on the number of the iteration at which it is induced. The vertical axis shows all the nogoods in such an order. Recall that in SysLS, given a current solution, we would like to know if a particular move will lead the current solution to a nogood previously kept. If so, the move will not be considered. It will be as if the nogood stands in the way of the search. The horizontal axis shows the number of the iteration at which the nogood stands in the way of the search.

\textsuperscript{3}http://www.mathworks.com/access/helpdesk/help/toolbox/stats/boxplot.html
Figure 5.3: Iterations at which all the nogoods are induced versus iterations at which some of the nogoods are in the way of the search for a single run of the SysLS for instance la01
Figure 5.4: Progress on the makespan versus number of iterations for instance la01
So if a nogood induced at the 16th iteration stands in the way of the search at iterations 16, 18, and 20, we would plot three points at (16, 16), (18, 16), and (20, 16) respectively. It is not difficult to observe that the nogoods represented as points in the figure roughly fit into a linear regression. This is not surprising because the nogoods are forcing the search into a new area of the search space as the search induces new nogoods along the way. On the other hand, the nogood points appear to be rather sparse because the heuristic is good enough to avoid visiting the same area of the search space repetitively.

So far it has been about using nogoods to diversify the search. Nogoods have been shown to be quite effective in driving a search out of local optima already visited, but how is the search doing on improving the makespan? To this end, we plotted the widely-used makespan-versus-iteration chart (Figure 5.4) in a slightly unconventional way. Rather than only plotting all the improving makespan values in the figure, we also plotted the trajectory of the search which goes through all the solutions ever visited. Figure 5.4 reflects the makespan values of all the solutions visited by the SysLS. Improving ones are marked by a square (□). The figure clearly shows that SysLS made good progress on the makespan value as well as in escaping local optima by accepting moves that lead to worse makespan values. When we compare Figure 5.3 and Figure 5.4, we can observe that during iterations 15 to 30 when the majority of the nogoods were induced and stood in the way of search, there were a considerable number of uphill moves taken by the search as reflected by the hill-shaped trajectory. Finally the optimal makespan value of 666 for instance la01 was reached in 35 iterations.

We think Figure 5.3 and 5.4 complement the information revealed by Figure 5.1 and 5.2 in that they help us to better understand what is happening during the search in addition to what we get at the end of it. Figure 5.4 also reveals the search trajectory. We further conjecture that a search with the guidance of a good heuristic should leave a trail that roughly fits into a linear regression. We plan to investigate this and to study how we can utilize this information to design better heuristics.
Chapter 6

Conclusions

6.1 Summary

In [10], the Systematic Local Search originally developed in [18] is extended for constraint optimization. Given the importance of hybrid search as a control mechanism for tackling combinatorial optimization problems, we believe that this research points in the right direction. SysLS synthesizes desirable aspects of systematic search and stochastic local search. It is capable of moving heuristically in the search space by following local gradients and also of achieving systematicity. By recording nogoods at local optima, SysLS will never be trapped in places already explored.

In this thesis, we defined a particular instance of SysLS for the job shop scheduling problem. A case study demonstrated how SysLS can be applied to constraint optimization problems, especially in terms of how to define a nogood in relation to the objective function to be optimized. For a JSSP, we defined the nogood induced at a local optimum as the longest path in the directed acyclic graph corresponding to the job shop schedule. Experimental results on a challenging set of JSSP benchmark instances showed that SysLS produces very good results. The results also demonstrate that SysLS frequently outperforms general local search methods such as simulated annealing and tabu search, and often compares favourably with some highly specialized heuristic methods for the JSSP. We also studied the runtime performance of SysLS and analyzed the nogood utilization during the search. Experimental results showed that SysLS has very good runtime performance and nogoods help SysLS to diversify.
6.2 Future work

We would like to conclude that SysLS has achieved initial yet notable success in making itself applicable in the domain of constraint optimization. Its future success depends on its flexibility and applicability to other combinatorial optimization problems. It would be ideal if SysLS could solve more complex problems, especially those in the scheduling domain, given that job shop scheduling represents only a simple class of scheduling problems in terms of problem complexity and side constraints. Further extension includes applying SysLS to other scheduling problems such as multi-capacity resource scheduling where a machine can process more than one operation at a time. This may again require novel but consistent nogood definition and we look forward to making SysLS an easy-to-use and widely applicable method for scheduling problems in general.

One objective of extending Systematic Local Search is to enrich this hybrid search method so that it can not only synthesize desirable aspects of both constructive search and local search, but also smoothly integrate a variety of heuristics which have been found to be very effective in other stochastic search methods. So far the work has made good progress yet this search method is far from being full-fledged and overarching. Several possible directions for further investigation come immediately to mind.

As we pointed out in Chapter 2, one interesting extension is to investigate a more comprehensive valuation of the current variable assignments that is more informative and heuristic. It is common practice that most local search methods use the objective function value, for example the makespan in the case of the JSSP, to guide the search. However getting trapped in local optima and having difficulty escaping local optima are the immediate consequences of lacking other means of directing the search. SysLS will be more flexible if it can admit a range of other criteria for evaluating the quality of the variable assignment and assessing the progress of the search. One such criteria is the similarity among the solutions generated, for example the hamming distance between job shop schedules as suggested in [40]. The big valley [3] conjecture suggests that similar solutions have similar objective function values. By measuring how similar solutions are, we may be able to tell whether or not the search is in a big valley where local optima are clustered and global optima may even exist. We believe that valuations like this will provide us with much more comprehensive information, leading to improvements in search efficiency and performance.

When it comes to improving the SysLS's runtime performance and preventing search
stagnation, we can propose at least two methods for further development. The first is to simply restart the search after observing that the search has become slow in making progress. As in [29], we did try to restart the SysLS from several very good solutions while keeping all the nogoods. However we received mixed results. Sometimes SysLS made moderate improvement after a restart but sometimes it failed to make any further improvement at all. We chose not to present the results here since the improvement is not consistent across the benchmark instances. We conjecture the reason may be that the best makespan found is already very close to the optimal, thus making further improvement extremely difficult. On the other hand, a restart did help us on a few instances, thus suggesting further investigation is worthwhile.

Adjusting the search heuristic is another possible avenue for future research. Inspired by the work in [5] which changes search strategy by allocating computational resources among a set of different algorithms to achieve the best overall performance, it may be useful to change and adjust the heuristics as soon as we discover that the search is losing momentum. To put it more precisely, we can think of our heuristics placed in SysLS as being dynamic instead of static. We should change the heuristic if it keeps making the same kind of variable assignment contrary to what the induced nogoods suggest. We believe that the results revealed by Figure 5.3 and Figure 5.4 make the first step towards a deeper understanding of the nogood surface. The next step requires us to dig into the variable assignment level. This work is rather preliminary and it is hard to draw conclusions at this point. However we aim to boost SysLS's runtime performance by designing multiple heuristics to guide the search so that it can be dynamically managed for better results as compared to a static setting, given the same amount of running time.
Appendix A

JSSP Glossary

Here we provide a glossary of terms that are related to the job shop scheduling problem. A job shop scheduling problem instance has a set $J$ of $n$ jobs $\{J_1, J_2, \ldots, J_n\}$ and a set $M$ of $m$ machines $\{M_1, M_2, \ldots, M_m\}$. Each job is composed of a number of operations and each operation is to be processed on a machine for an uninterrupted duration. In this thesis, we are only looking at the instances where the number of operations contained by each job is exactly the same as the total number of machines and each operation is to be processed on a different machine. Therefore the number of operations in total is $n \times m$ since there are $n$ jobs and each job has $m$ operations.

**critical operation** A critical operation is an operation that cannot delay its execution (i.e. start execution later than its earliest start time) without delaying the schedule (i.e. increasing the makespan).

**critical path** A critical path consists of a sequence of operations in which an operation’s finish time is exactly the next operation’s start time except for the last operation in the sequence. The sequence contains only critical operations. The last operation’s finish time equals to the value of the makespan. Several critical paths may exist in a schedule.

**duration** Every operation has a fixed amount of time for its execution and this time is called duration.

**earliest start time** The earliest start time of an operation is the time at which either its job predecessor finishes its execution, if it has one, or its machine predecessor finishes
its execution, if it has one, and whichever is greater. The very first operation that is executed in a schedule has an earliest start time of zero.

**finish time** An operation's finish time equals to the time of its earliest start time plus its duration.

**job** A job is composed of a number of operations. It specifies the processing ordering among its operations.

**job predecessor** An operation's job predecessor is the operation that is placed before it in the job. The first operation of a job does not have a job predecessor.

**job successor** An operation's job successor is the operation that is placed after it in the job. The last operation of a job does not have a job successor.

**machine** A machine is responsible for processing operations. At any given time, a machine is either idling or processing one operation of a job.

**machine predecessor** An operation's machine predecessor is the operation that is scheduled to be executed before it. The first operation to be executed on a machine does not have a machine predecessor.

**machine successor** An operation's machine successor is the operation that is scheduled to be executed after it. The last operation to be executed on a machine does not have a machine successor.

**makespan** If the schedule starts at time zero, then the makespan value is the time at which the last operation finishes its execution if every operation starts its execution at its earliest start time.

**operation** An operation must be processed on a machine for a fixed duration. It cannot be interrupted during its execution.

**schedule** A schedule is obtained after the precedence ordering among all the operations to be processed on the same machine is imposed and every operation starts its execution at its earliest start time. A feasible schedule must have no cycles.
Bibliography


