Repair-Based Scheduling

Repair-Based Scheduling

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. dr. A.C. Nieuwenhuijzen Kruseman,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op woensdag 18 juni 2003 om 14:00 uur

door

Yong-Ping Ran

SIKS Dissertation Series No. 2003-8

NWO The research reported in this thesis was funded by the Netherlands Organization for Scientific Research (NWO)

ISBN XXXX

Contents

Preface

The research presented in this thesis was funded by the Netherlands Organization for Scientific Research (NWO) and performed at the Institute for Knowledge and Agent Technology (IKAT), Universiteit Maastricht, the Netherlands.

I would like to thank my supervisor, Jaap van den Herik, for his continuous support, guidance, and encouragement. His teaching in scientific research and writing is greatly appreciated. Without his enthusiastic help and intervention, this thesis would never have appeared.

Sincere thanks go to Nico Roos, my daily supervisor. He carefully guided me along the obstacles in the domain and showed me how to perform an in-depth analysis. His advice and inspiration made this study progress. I am grateful for his patient guidance, fruitful advice, and constructive criticism.

The guidance of Ida Sprinkhuizen-Kuyper has been invaluable for the proofs of the theorems in Chapters 4 and 5. I gratefully recognize her willingness to read the manuscript and to scrutinize the proofs. Her insisting on clarity and correctness is engraved on my memory.

Moreover, I would like to thank Joop Verbeek, Evgueni Smirnov, Rens Kortmann, Frank Claus, Peter Geurtz, Levente Kocsis, and Mark Winands for being with me in difficult times of the research and thesis writing. Their encouragement and assistance made my life and work in Maastricht much easier.

Furthermore, I would like to acknowledge Jeroen Donkers and Floris Wiesman for sharing their experience in programming and LaTeX. This facilitated the experimental study presented in the thesis as well as the preparation of the final manuscript.

Many people provided assistance or valuable advice. I thank, next to those already mentioned, Jos Uiterwijk, Jan van Zanten, Eric Postma, Leo Coolen, Arno Sprinkhuizen, Erik van der Werf, Paul van der Krogt, Rui Ferreira, Jacques Lenting, Sandro Etalle, Paul Vogt, Menthy Willems, and Robert Plompen for their attentiveness.

In addition to all these persons, I would like to express my gratitude towards Joke Hellemons, Marlies van der Mee, Martine Tiessen, Hazel den Hoed, and Anjes Schreurs. Over the years their kindness and support in administrative matters facilitated my work and study in Maastricht. Special thanks go to Marlies van der

Mee for her organization of the Ph.D. procedure and defence. I am grateful to Joke Hellemons for her continuous help and advice about living in Maastricht.

During my stay in Maastricht I have very much appreciated the support of my friends and country fellows who lived or studied in the Netherlands, too. They made me happy in my leisure time.

With much respect and pleasure, I wish to thank my parents for their restless worrying about me. I am greatly indebted to my wife, Ge Rong, for her love, endurance, encouragement, and understanding, as well as for her sacrificing her own career to support me during my study. Finally, I am indebted to my daughter, Yao Ran, for her tolerance to my frequently ignoring of her requirements when I was too devoted to my study.

Acknowledgements The research reported in this thesis has been carried out under auspices of NWO, the Netherlands Organization for Scientific Research, and SIKS, the Dutch Research School for Information and Knowledge Systems.

Yong-Ping Ran Maastricht, April 2003

Chapter 1

Introduction

In this thesis we investigate repair-based approaches for handling dynamic Constraint Satisfaction Problems (CSPs) and reactive Job Shop Scheduling Problems (JSSPs) in the field of Artificial Intelligence (AI). In this chapter we present the contours of these problems and outline our research approaches. The principal problems of the repair-based approaches are formulated and emphasized in the thesis's problem statement, which determine the lines to be followed in the subsequent chapters.

1.1 Background

A large number of problems in AI and other areas of computer science can be viewed as special cases of constraint-satisfaction problems (Kumar [?]). A few examples are machine vision, belief maintenance, scheduling and planning, temporal reasoning, graph problems, circuit design, diagnostic reasoning, data mining [?] and access control to multimedia storage devices [?].

The first part of this research concentrates on developing repair-based approaches for Dynamic Constraint Satisfaction Problems (DCSPs); the second part on developing Repair-Based Scheduling approaches for JSSPs.

1.1.1 CSPs and Dynamic CSPs

Constraint Satisfaction Problem A general CSP consists of a set of variables, a finite set of possible values for each variable (variable domain), and a set of constraints restricting the values that the variables can simultaneously take. A solution to a CSP is a complete assignment that assigns to each variable a value from its domain in such a way that all constraints between the variables are satisfied. Depending on the requirements of an application, the task for solving a CSP can be classified into finding: (1) just one solution, with no preference to which one; (2) all solutions; and (3) a particular solution according to prefixed domain conditions as formulated by functions that can be assessed straightforwardly. However, not every CSP is solvable. In the latter case, the solver is required to prove that no solution exists for the given CSP. Subsequently, the solver can stop the work, or relax some constraints to find a solution for the new case under investigation. The solver may attempt to obtain a partial solution that satisfies as many of the constraints as possible. A CSP can be considered as a declarative description of a combinatorial problem in which a solution results from taking a whole series of interdependent choices, i.e., the correctness of a given choice is usually not apparent until a number of other choices have been made, which may in turn depend on further choices.

The question whether a CSP has a solution is NP-complete, meaning that a polynomial algorithm to solve all CSPs is unlikely to exist. In order to seek for a complete assignment that satisfies all constraints in a CSP, search methods are widely applied. However, search can be very expensive, i.e., it may require a large amount of computation time. During the past two decades, a large number of efficient search strategies and techniques have been developed to handle the CSPs in various application domains [**?**, **?**, **?**, **?**]. These techniques include proper variable and value selection heuristics for guiding the search process[**?**, **?**, **?**]; constraint-propagation for pruning the search space and making the search easier and more efficient [**?**, **?**, **?**, **?**, **?**]; and intelligent backtracking for handling adequately the dead-ends in the search processes[**?**, **?**, **?**].

Optimization Many practical problems can easily be formulated as a CSP. In most cases finding a solution to a CSP requires domain-specific knowledge, but general methods for solving CSPs are applicable in many situations.

In real-world applications it is often of the utmost importance whether the solution found reaches the predefined objective. Then, seeking for a particular solution is necessary. Normally, the predefined objective is given by a mathematical function that will be regarded as an additional constraint to the CSP. The requirement then is to find a solution that optimally meets the objective function, rather than to find an arbitrary solution. These problems are called Constraint Optimization Problems (COPs). An optimization problem is called *NP-hard* if the corresponding decision problem is NP-complete. Straightforwardly, COPs are NP-hard problems.

In general, an optimization poses two tasks: (1) searching for a feasible solution, and (2) proving that the result is optimal. A naive but sometimes costly approach to obtain an optimal solution is to find all solutions for the corresponding CSP and then compare the solutions with respect to the objective function. To avoid looking for all solutions, sometimes domain knowledge can be used to prune parts of the search space that do not contain solutions better than the best solution ('best' according to the objective function) found so far. This strategy can be realized by incorporating a simple inference (*branch and bound* algorithms, see [**?**]) into the search processes.

The time costs to find an optimal solution is usually higher than that to find an arbitrary solution. So, efficiency becomes a prominent issue in developing optimization algorithms. In particular, in some applications, where the environment changes dynamically, an optimal solution at one point may become suboptimal very soon. The decisions derived from the solution could be useless if they come too late. Hence, giving up completeness for speed is crucial in these cases. The weakening of the optimality criteria, i.e., relaxing an additional constraint specified by the objective function and obtaining a near-optimal solution, is then considered acceptable and sufficient. For a hard optimization problem, one can crack (decompose) it by solving a reasonable number of easy problems that are collectively equivalent to the original problem [**?**, **?**]. Each of them corresponds to a new problem which may be much easier to solve than the original problem. When no alternative methods are available, the stochastic search techniques could be used to solve a CSP and a COP. A stochastic search algorithm moves from one point to another in the search space in a non-deterministic manner, guided by heuristics. The next move in the search space is partly determined by the outcome of the previous move. Generally speaking, a stochastic search is incomplete. Thus, the obtained solution is often near-optimal for a COP.

Dynamic CSPs The technologies developed for solving CSPs (COPs) have been successfully used to solve combinatorial (optimization) problems due to their combination of high-level modeling, constraint propagation and facilities for controlling search behavior. However, the physical world is a highly dynamic environment. The set of constraints in a CSP which models a real-world problem may change over time. For instance, a machine may break down at one time in a job shop. The resource constraint which specifies the availability of the machine thus changes. Such a change may result in a new CSP. The sequence of such CSPs is called a Dynamic CSP (DCSP) [**?**]. The constraint change of a CSP may be a restriction (a new constraint is imposed on a subset of variables), a relaxation (a constraint is removed from the CSP) or a combination of restriction and relaxation.

In case of the restriction or the combination of relaxation and restriction, a valid solution of one CSP may not be a solution of the next CSP invoked by the new situation. Hence, a new solution must be generated for that CSP. In solving such CSPs, it is always possible to treat them as independent CSPs and adopt the normal CSP solution methods. However, any solution obtained in this way might be quite different from a previous one. In practical situations, large differences between successive solutions are often undesirable. For instance, airline companies are continuously faced with disruption problems which derive from severe weather patterns and unexpected aircraft or personnel failures. When a disruption occurs, new aircraft routings are needed. Changes to aircraft routings inconvenience passengers and affect crew schedules, gate assignments and maintenance. Reassigning more than a few aircrafts when a single plane is grounded is unacceptable for most airlines. Consequently, dealing with disruption in a way that results in a minimal difference from the original aircraft routings is very desirable [?]. So, in these cases we practically face the challenges to solve a DCSP with a *solution-maintenance* objective.

In dealing with these challenges, some methods were proposed [?, ?] to generate solutions that are expected to remain valid after constraint changes in a CSP. These approaches are based on the assumption that the successive changes in a DCSP are temporary and tractable. However, when unexpected events occur, for instance a mechanical problem (a machine breakdown) or an employee illness, the assumptions may no longer be well-founded. Since it is quite complicated to track down the changes directly in the cases of constraint addition, those approaches may become too costly to be viable [?]. Verfaillie and Schiex [?] proposed an alternative algorithm by reusing the solution of a previous CSP to find a solution for the new CSP. The solution maintenance was made by fixing the assignment of a set of variables in the solution of the previous CSP, and then finding a partial assignment for the rest of variables that are consistent with the fixed assignments. However, it turned out to be hard to predict which assignment of variables should be fixed in a dynamic environment. Moreover, carrying out the solution maintenance by fixing the assignment of a set of variables is rather limited in some specific application domains. In other application domains such as generating a new aircraft routing, the objective may be to maintain as many as possible variable assignments equal to the old ones. To meet these application demands, the first part of our research focuses on exploring new approaches for solving such solution-maintenance problems in a general DCSP. From the solution-maintenance point of view, the new approach carries out the tasks of repairing/revising some assignments in the original solution. So, it is purposedly called a repair-based approach in solving DCSPs.

1.1.2 Repair-Based Scheduling for JSSPs

In the second part of this research, we concentrate on developing Repair-Based Scheduling approaches to deal with the unexpected events occurring in a manufacturing system.

Job Shop Scheduling Problem (JSSP) Scheduling is motivated by questions that arise in various domains of application ranging from manufacturing, computer control, train and airline planning to generally all situations in which scarce resources have to be allocated to activities (tasks) over time [?]. In this thesis, we focus on the applications in a manufacturing environment, where scheduling is the assignment of operations of jobs to machines for a given period of time on the job shop floor. The corresponding scheduling problem is called Job Shop Scheduling Problem (JSSP).

The JSSP considered in our research is a standard JSSP which consists of a number of jobs and a limited number of resources. Each job consists of a series of operations which are subjected to a linear order (often called precedence constraint). Each operation requires a resource, often denoted as a machine, for processing it.

The machine can be allocated to at most one operation at a time (often called capacity constraint). Moreover, a machine is allocated to an operation for a certain processing time period (from the start of processing the operation till the end of processing the operation, i.e., non preemptive scheduling). A solution to a JSSP is a complete assignment that assigns to each operation a start time in such a way that the precedence and capacity constraints in the JSSP are satisfied. The standard JSSP is not only NP-hard [?] but also one of the computationally more difficult combinatorial optimization problems [?].

Operation Research and AI Operations Research analysts and engineers have been pursuing solutions to these problems for more than 35 years, with varying degrees of success. While they are difficult to solve, JSSPs are among the most important problems since they have an impact on the ability of manufactures to meet customer demands and make a profit. They have an impact, too, on the ability of autonomous systems optimizing their operations, the deployment of intelligent systems, and the optimizations of communications systems. Hence, effective job shop scheduling may make a major contribution to the competitive power in serving customers, and in utilizing the assets and resources of a company.

Existing approaches for solving JSSPs range from the combinatorial optimization techniques of operation research to AI approaches. However, the approaches purely developed in the field of operation research (OR) have generally proved unsuccessful [?]. The currently known best results (with respect to time and solution quality) are achieved by hybrid approximation algorithms [?] that combine OR and AI technologies. As pointed out by Fox [?], practical scheduling problems are affected by a large number of domain-specific constraints most of which are not readily expressible in conventional OR oriented forms of representation. Nevertheless, AI approaches for JSSPs represent an important step towards the required expressiveness, flexibility and responsiveness in handling realistic scheduling problems. These approaches provide a rich representational language for describing, propagating and if necessary relaxing the many constraints that apply to schedules in live environments, and multiple search strategies to assist in finding a solution.

In order to find a feasible schedule to a JSSP, AI approaches generally model the JSSP as a CSP and construct a search tree which will be explored in a depth-first or iterative-deepening mode depending on the application. Various methods, among them the constraint-directed search techniques proposed by Fox [?] and the randomized strategy developed by Nuijten [?, ?], can be applied to reduce the size of the search tree and to achieve a feasible schedule with the lowest effort.

Reactive Scheduling Generating a schedule before its execution is called predictive scheduling [?]. As we mentioned in dealing with DCSPs, the physical world is a highly dynamic environment which comprises a large number of uncertainties. In a manufacturing shop floor, fast changing customer demands may evoke external, market-driven uncertainties, while unplanned shop floor contingency such as a machine breakdown, a late delivery, an operator being absent, etc., comprise internal manufacturing uncertainties. Most of these uncertainties are unforeseeable at the moment of generating a predictive schedule. Once an unexpected event occurs, the predictive schedules may no longer be valid. Thus, they need to be revised continuously for the management of changes at the shop floor level. Revising the predictive schedule as unexpected events force changes is called *reactive scheduling* [?].

There are a number of other definitions for reactive scheduling in the scheduling literature [?, ?, ?, ?]. Although these definitions are somewhat general in nature, they all emphasize the fast-changing environment, which demands immediate decision at shop floor level as a reactive response to contingencies. As pointed out in [?], reactive scheduling problems, like their predictive counterparts, are typically a domain for which no polynomial-time solution algorithm is known. Computation times are usually highly variable and unpredictable. So, giving up completeness and optimality criteria in reactive scheduling, i.e., relaxing the constraint specified by the objective function, is therefore considered necessary and acceptable.

In the past decade, many reactive scheduling systems have been developed [?, ?, ?, ?]. Although these systems emphasized the importance of reducing the disruption to the original (predictive) schedule when generating a new schedule, they were primarily designed to balance the objective of minimal disruption with the traditional optimization objectives (such as minimizing the make-span, work-in-process inventory, mean tardiness of jobs etc). In a real-world scheduling environment, minimizing the disruption to the original schedule may need particular attention and be treated separately from the traditional optimization objectives. The reason is that the original (predictive) schedule represents an investment in planned resources, i.e., an allocation of machines and people. By executing the original schedule a large number of interdependent processes has been put into motion. If an unexpected event occurs, keeping as much as possible the continuity of those processes is of vital importance for reaching an agreement among all parties affected by the changes on the original schedule.

Minimizing the disruption to the original (predictive) schedule in reactive scheduling provides a rich context of Repair-Based Scheduling, which is interpreted as generating a new schedule that has a minimal difference with the original (predictive) schedule. It means that Repair-Based Scheduling must achieve two objectives simultaneously: (1) the modification of the original (predictive) schedule should be as minimal as possible; (2) the time required to get or validate a new schedule should not exceed the time window in which the schedule must be applicable. The difference between a new schedule and the original schedule may be explicitly described by some objective functions. The time required by the Repair-Based Scheduling system may be measured by the run time (in CPUs) of the corresponding algorithms. The new JSSP problems caused by unexpected events are called repair-needed JSSPs in the context of Repair-Based Scheduling.

Since the reactive scheduling systems currently in use do not uniquely pursue the objective that Repair-Based Scheduling seeks, we intend to develop an efficient and robust Repair-Based Scheduling system to meet the original goal. Therefore, we deal with the Repair-Based Scheduling in the field of AI. Hence, the system will be built under the fundamental framework of AI scheduling: constraint-directed search.

1.2 Problem statement

As described in the previous section, a valid solution of one CSP in a DCSP may not be a solution of the next CSP when an additional constraint is added. Hence, a new solution must be generated for that CSP. Our goal is to solve, as efficiently as possible, such a DCSP with a solution-maintenance objective. To meet the application demands, we need to explore repair-based approaches for solving a general DCSP.

In a Repair-Based Scheduling approach, a new schedule is required to cope with the changes caused by an unexpected event (a machine breakdown) occurring in a job shop. To generate a new schedule that has a minimal difference with the original (predictive) schedule, we intend to use the constraint-directed search as the foundation to develop efficient, robust solution methods.

From the above, it is clear that the general research problem is twofold:

- Is it possible to develop new methods that adequately solve DCSPs?
- Is it possible to develop new methods that adequately solve repair-needed JSSPs?

This thesis is an attempt to address the general research problem. The subsequent chapters will describe our approaches in detail. The chapters are organized as follows.

1.3 Thesis outline

Chapter 2: Constraint Satisfaction Problems This chapter starts with giving a formal definition of a CSP. Then, it shows that solving a CSP is a NP-complete problem. The search methods and techniques for solving CSPs, which include variable and value selection heuristics for guiding search, constraint-propagation techniques for pruning the search space and dead-end handling techniques, are discussed subsequently. Finally, the optimization problem of CSPs (COPs) and Dynamic CSPs (DCSPs) are investigated.

Chapter 3: Job Shop Scheduling The goal of this chapter is to provide an overview of Job Shop Scheduling. First, a formal definition and a useful CSP model of a JSSP are presented. Then, a number of important notations in job shop scheduling which will be adopted in the successive chapters are introduced. Furthermore, some powerful methods which have successfully been used for solving JSSPs in the AI community are discussed and investigated. These methods include constraint-based scheduling approaches, local search algorithms and genetic algorithms. In particular attention is paid to the constraint-based scheduling which uses a constraint-directed search framework for solving JSSPs. Subsequently, the algorithms developed under the framework in generating a job shop schedule (with an optimal or a near-optimal make-span) and a set of constraint-propagation techniques in JSSP domain are briefly described. Finally, the concepts of predictive and reactive scheduling are brought in. The rescheduling and incremental repair strategies for reactive scheduling are briefly discussed and a number of typical reactive-scheduling systems are mentioned.

Chapter 4: A repair-based method for DCSPs In this chapter, we propose three repair-based methods for solving DCSPs, which include a complete repair-based algorithm (RB-AC) and two approximate algorithms (BS and RS). First, the necessity of finding a minimal-number of assignment-change solution for a CSP is clarified as the objective of the repair-based approaches. Then, the idea and methodology behind the methods are explained. Subsequently, a complete repair-based algorithm (RB-AC) which combines local search and constraint-propagation techniques is proposed. The termination, correctness, completeness and optimality of the RB-AC is proved. Following the analysis of the time complexity of RB-AC, two approximate algorithms which are developed to obtain a near minimal-number of assignment-change solution are proposed. Through empirical studies, we conclude that the approximate algorithm RS outperforms BS, and finally introduce the best parameter combinations in RS.

Chapter 5: Repair-Based Scheduling This chapter presents explorative studies for Repair-Based Scheduling in the field of AI. First, the motivation in conducting a repair for an existing schedule is clarified. Then, we discuss the following four issues: (1) why is an innovative Repair-Based Scheduling approach needed? (2) what kind of model modification should be made to the original JSSP model to arrive at an adequate solution? (3) what is the objective that must be achieved in Repair-Based Scheduling? and (4) what minimal perturbation function is appropriate for describing the objective? After that, we make a thorough analysis to an unexpected event (e.g., a machine breakdown) occurring in a job shop and build a new CSP model for the repair-needed JSSP. Therefore, we identify which operations need to participate in repair and what constraints should be added to the original constraint set. Subsequently, we outline a novel Repair-Based Scheduling algorithm (RBS) which is developed under the framework of constraint-directed search. The main ideas, related techniques and the pseudo codes of the RBS are presented and illustrated. Finally, the procedures which form the main part of an innovative heuristic (semi-randomized heuristic) and the different design choices of RBS are presented. A new constraint-propagation technique which impose the optimization needs on the operations' start-time domains and thus help to prune the search space is developed in the successive sections. Finally, an algorithm to implement this technique is shown.

Chapter 6: Performance evaluation In this chapter, we report the performance of our Repair-Based Scheduling approaches in solving repair-needed JSSP instances. First, we explain how to generate a repair-needed JSSP instance (caused by a machine-breakdown) from a standard JSSP instance. The important parameters of such a machine-breakdown JSSP instance are discussed and determined. Subsequently, we present the experimental results by using RBS to solve simple machine-breakdown instances. After that, the different design choices of RBS are evaluated in solving simple, moderate and difficult machine-breakdown instances. The experimental results of running these algorithms for different problems are compared and discussed. Then, the representative of our Repair-Based Scheduling approaches is selected. The important parameters in RBS which include the number of restarts, the backtracking factors and the probability partition factors in the extended semi-randomized heuristic are dealt with different values in a variety of experiments. The efficiency and optimality impact of these parameters to RBS are clearly exhibited by the experimental results. The experimental results of other alternative operation selection heuristics are also presented to compare with the schedules generated by RBS. Finally, the issues about the comparison with other existed reactive-scheduling systems are discussed.

Chapter 7: Concluding remarks The final chapter concludes our research with summarized answers to the problem statement formulated in Section 1.2. Furthermore, the contributions of our research covered by this thesis are outlined and a short discussion on future research is given.

Chapter 2

Constraint Satisfaction Problems

This chapter presents the formal definitions of a Constraint Satisfaction Problem (CSP) and a Dynamic CSP (DCSP) (Section 2.1), the relevant issues of the complexity theory (Section 2.2), and the important notions that are concerned with solving a CSP (Section 2.3). For instance, we review tree-search algorithms, constraint propagation, and heuristics for variable and value selection. Finally, in section 2.4, solution methods and the solution-maintenance problem for a DCSP are discussed.

2.1 Basic notations and formal definitions

A CSP consists of a finite set of variables, a finite set of possible values for each variable (its domain), and a finite set of constraints restricting the values that the variables can simultaneously take. A solution to a CSP is a complete assignment that assigns to each variable a value from its domain in such a way that all constraints between the variables are satisfied. In practice, a large number of problems can be molded as CSPs. Consequently, the methods and techniques developed for solving CSPs can be applied to handle these problems. Overviews on CSPs can be found in [**?**, **?**, **?**].

2.1.1 Unary, binary and general CSPs

In this subsection, we formally define a variable's domain, a CSP, a partial assignment, a complete assignment, a solution of a CSP and the notion satisfiable (consistent).

Definition 1 *The domain of a variable is a set of all values that can be assigned to the variable. If v is a variable, we use D_v to denote the domain of v.*

The domain of a variable can be numbers, boolean values, enumerated type of objects etc. When the domain of a variable contains numbers only, the variables are called *numerical variables*. Moreover, the domain of a variable can be a finite set of values or an infinite set. In this thesis, we focus on *numerical variables* with a *finite set of values.*

Definition 2 *A constraint satisfaction problem (V, D, C) involves a set $V = \{v_1, v_2, \cdots, v_n\}$ of n variables, a set of finite domains $D = \{D_{v_1}, D_{v_2}, \cdots, D_{v_n}\}$ and a finite set $C = \{c_{v_{i_1},\cdots,v_{i_k}} | 1 \le i_1 < \cdots < i_k \le n, k \le n\}$ of constraints. A constraint*

$$c_{v_{i_1},\cdots,v_{i_k}} : D_{v_{i_1}} \times \cdots \times D_{v_{i_k}} \rightarrow \{true, false\}$$

is a polynomial-time computable mapping onto true or false for an instance of $D_{v_{i_1}} \times \cdots \times D_{v_{i_k}}$.

For a constraint $c_{v_{i_1},\cdots,v_{i_k}} \in C$, $\{v_{i_1}, \cdots, v_{i_k}\}$ is the set of variables involved in constraint $c_{v_{i_1},\cdots,v_{i_k}}$, where $1 \le i_1 < \cdots < i_k \le n, k \le n$. A constraint may involve an arbitrary number $k(k \le n)$ of variables. Typically, if a constraint involves one variable, where $k = 1$, the constraint is called a *unary* constraint. If a constraint involves two variables, where $k = 2$, the constraint is called a *binary* constraint. A CSP with constraints not limited to unary and binary constraints is called a general CSP. Nudel [?] proved that a general CSP can be reformulated as a binary CSP. Hence, in the CSP research, a great deal of attention is paid to binary CSPs. An instance of a binary CSP can be viewed as a *graph*(or *constraint network/graph*), where the nodes represent variables and the edge (v_1, v_2) represents the constraint c_{v_1,v_2}. However, the graph interpretation of a general CSP is not so obvious. For this reason, Dechter and Pearl [?] introduced the notion *constraint hypergraph* for general CSPs.

A variable in a CSP(V, D, C) can be assigned a value from its domain. Below, we formally define a partial assignment, a complete assignment, a solution of a CSP, and the notion satisfiable (consistent).

Definition 3 *A partial assignment for a CSP (V, D, C) is a function a' that assigns values to variables in a subset V' of V, i.e., $\forall\ v' \in V'(V' \subseteq V), a'(v') \in D_{v'}$. A complete assignment $a^\star$ is a function that assigns a value to every variable appearing in the CSP, i.e., $\forall\ v \in V, a^\star(v) \in D_v$.*

Definition 4 *For a CSP (V, D, C), let a' be a partial assignment on $V'(V' \subseteq V)$. a' satisfies the constraint $c_{v_{i_1},\cdots,v_{i_k}} (c_{v_{i_1},\cdots,v_{i_k}} \in C, \{v_{i_1}, \cdots, v_{i_k}\} \subseteq V')$ iff*

$$c_{v_{i_1},\cdots,v_{i_k}}(a'(v_{i_1}), \cdots, a'(v_{i_k})) = true.$$

Definition 5 *For a CSP (V, D, C), let $a^\star$ be a complete assignment. Then $a^\star$ is a solution for the CSP iff for each $c_{v_{i_1},\cdots,v_{i_k}} \in C$, $a^\star$ satisfies the constraint $c_{v_{i_1},\cdots,v_{i_k}}$.*

Definition 6 *A CSP is called* satisfiable (consistent) *if a solution exists.*

2.1.2 Tightness of a CSP

CSPs can be adequately characterized by their *tightness*, which could be measured under the following definitions (adapted from [**?**]). Below we define the tightness of a constraint and the tightness of a CSP.

Definition 7 *The tightness of a constraint $c_{v_{i_1},\cdots,v_{i_k}}$ is measured by the number of partial assignments satisfying $c_{v_{i_1},\cdots,v_{i_k}}$ over the number of all partial assignments on $\{v_{i_1}, \cdots, v_{i_k}\}$:*

For a CSP(V,D,C): $\forall c_{v_{i_1},\cdots,v_{i_k}} \in C$:

tightness$(c_{v_{i_1},\cdots,v_{i_k}}; (V, D, C)) \equiv \frac{S}{T}$

where:

$S = \|a \; |c_{v_{i_1},\cdots,v_{i_k}}(a(v_{i_1}), \cdots, a(v_{i_k})) = true\|$

$T =$ *maximum number of assignments for* $v_{i_1}, \cdots, v_{i_k} = \prod_{j=1}^{k} \|D_{v_{i_j}}\|$.

Definition 8 *The tightness of a CSP is measured by the number of solutions over the number of all distinct complete assignments :*

For a CSP(V,D,C): tightness$((V, D, C)) \equiv \frac{\|S\|}{\prod_{\forall v \in V} \|D_v\|}$

where $S =$ the set of solutions.

Tightness is a relative measure. Some CSP solving techniques are suitable for tighter problems, while others are suitable for looser problems. In order to evaluate a CSP solving algorithm as precise as possible, the CSP instances are usually considered for different CSPs and various constraint tightnesses. The typical CSP instances that can be used for this purpose are randomly generated binary CSPs denoted by a four-tuple (n, d, p_1, p_2), in which n denotes the number of variables and d the domain size of each variable; p_1 and p_2 are two probabilities. Probability p_1 represents the probability that a constraint exists between two variables and probability p_2 represents the conditional probability that a pair of values in the domains of two variables satisfies the constraint between them. So, when p_1 takes values from 0 to 1, the generated CSP instances range from the loosest CSPs to the tightest CSPs. When p_1 is fixed and p_2 takes values from 0 to 1, the generated constraints between variables range from the tightest constraints to the loosest constraints.

The suitability of the specific techniques for different tightnesses of CSPs and constraints is discussed in Subsection 2.3.3 and in Chapter 4 (in the description of the corresponding techniques).

2.2 Complexity of CSP

In practice, some CSPs are easier to solve than others. Owing to complexity theory which relies on a mathematical exploration of general computational problems, CSP problems can be classified into "easy problems" and "hard problems". Below we briefly discuss the notions algorithm and efficiency (Subsection 2.2.1), NP and NP-completeness (Subsection 2.2.2), and the main theorem of CSP that a CSP is an NP-complete problem (Subsection 2.2.3). In dealing with complexity theory and its theorem we mainly refer to William et al. [?] and Brucker [?].

2.2.1 Algorithms and efficiency

When we use a computer to solve a problem, we expect that the computer produces an output $f(x)$ for each input x in some given domain. Although it is well known that a computer solves the problem by a list of instructions, called an algorithm, we nevertheless define the notion since it is the basis of complexity theory.

Definition 9 *An algorithm is a finite list of instructions to solve a problem.*

For a precise description, a Turing machine is commonly used as a mathematical model of an algorithm. For our purpose, it is sufficient to think of a computer program written in some standard programming language as a model of an algorithm.

The efficiency of an algorithm is characterized by the performance of an algorithm with respect to its computational time. The latter is measured by an upper bound $T(n)$ which is related to the number of steps that the algorithm takes on any input x with $n = |x|$, where $|x|$ denotes the length of some encoding of x. In most cases, it is difficult to calculate the precise form of T. For these reasons, the T is replaced by its asymptotic order. Therefore, we say that $T(n) \in O(g(n))$ (g is a function) if there exist two constants c and n_0 with $c > 0$ and n_0 a nonnegative integer such that $T(n) \leq cg(n)$ holds for all integers $n \geq n_0$. Thus, instead of saying that the computational complexity is bounded by $8n^3 + 15n^2 + 6n + 7$, we simply say that it is $O(n^3)$.

Definition 10 *A problem is called polynomially solvable if there exists a polynomial p such that $T(|x|) \in O(p(|x|))$ holds for all inputs x of the problem, i.e., if there is a integer k such that $T(|x|) \in O(|x|^k)$.*

Somewhat informally and phrased beyond mathematical rigor we may state: a problem is a question or a task. So, problems can be categorized into two types: problems that can be answered by *yes* or *no*, and those that ask you to find a certain object. The first type of problems is called *decision* problems. The second type of problems can be easily transformed into equivalent decision problems by rephrasing the conditions in the task. For instance, a scheduling problem can be transformed into a decision problem by defining a threshold K for the corresponding objective function f. Such a decision problem then reads: does there exist a feasible schedule S such that $f(S) \leq K$?

2.2.2 NP and NP-completeness

The class of problems solvable in polynomial time is usually denoted by P. The class NP is a class of decision problems where each "yes" input x has a certificate y, such that $|y|$ is bounded by a polynomial in $|x|$ and there is a polynomial algorithm to verify that y is a valid certificate for x. The letters NP stand for *nondeterministic polynomial time*. Clearly, $P \subseteq NP$ and NP is assumed to be a much larger class than the class P.

Within the class NP there are *NP-complete* problems. These are the hardest problems in the class NP. Below we define NP-complete problems, but we start defining the concept of a polynomial-time reduction.

Definition 11 *Let Q, R be two decision problems, Q polynomially reduces to R if there exists a polynomial-time function g that transforms inputs for Q into inputs for R such that x is a 'yes'-input for Q iff $g(x)$ is a 'yes'-input for R.*

Definition 12 *A decision problem $Q \in NP$ is called NP-complete if for each decision problem $Q' \in NP$ there exists a polynomial-time reduction of Q' to Q.*

Obviously, if $Q \in P$ and there exists a polynomial-time reduction of Q' to Q, then $Q' \in P$. This implies that if one NP-complete problem can be solved in polynomial time, then each problem in NP can be solved in polynomial time. Despite considerable research effort, so far no polynomial-time algorithm is found for any of the NP-complete problems. Thus, it is widely accepted that any algorithm that solves each instance of an NP-complete problem requires *super-polynomial* time. We will deal with such problems and corresponding algorithms in the next subsection.

2.2.3 The main theorem

The size of an instance of a CSP is characterized by the number of variables, i.e., each instance of the CSP can be encoded by $p(n)$, where p is a polynomial and n is the number of variables. This follows from the fact that the size of the domains are polynomial in n, the number of constraints is polynomial in n, and each constraint is polynomially computable. The complexity of a CSP is given by the following theorem.

Theorem 1 *The decision version of a CSP, i.e., the question whether the CSP has a solution, is an NP-complete problem.*

Proof Observe that a CSP belongs to the class NP, because for a given assignment a, all constraints can be checked in polynomial time. So, we can verify in polynomial time the correctness of a solution of a CSP instance.

Next we prove that an instance of 3-SAT (satisfiability) problem $\langle \mathcal{V}, \mathcal{C} \rangle$ can be polynomially reduced to a CSP, where:

- $\mathcal{V}$ is a set of boolean variables;
- $\mathcal{C}$ is a set of clauses: $cl \subseteq \mathcal{L} \times \mathcal{L} \times \mathcal{L}$,
  $\mathcal{L} = \{\neg v \mid v \in \mathcal{V}\} \cup \mathcal{V}$.

Let $f : \mathcal{V} \to \mathcal{W}$ be a polynomially bijective function that maps the variables of $\mathcal{V}$ onto a set of new variables $\mathcal{W}$ ($\mathcal{W} = \{f(v) \mid v \in \mathcal{V}\}$).

We reduce the 3-SAT problem to the following CSP:

$$\langle \mathcal{W}, \{\{0,1\}_w \mid w \in \mathcal{W}\}, \mathcal{C}' \rangle.$$

$cl = (\ell_1, \ell_2, \ell_3) \in \mathcal{C}$ if and only if

$$c_{w_1,w_2,w_3} \in \mathcal{C}';$$

$$w_i = \begin{cases} f(\ell_i) & \text{if } \ell_i \in \mathcal{V} \\ f(v) & \text{if } \ell_i = \neg v \text{ and } v \in \mathcal{V}; \end{cases}$$

$$t_i(x) = \begin{cases} x & \text{if } \ell_i \in \mathcal{V} \\ 1 - x & \text{if } \ell_i = \neg v \text{ and } v \in \mathcal{V}; \end{cases}$$

$$c_{w_1,w_2,w_3}(x,y,z) = \begin{cases} \mathit{true} & \text{if } t_1(x) + t_2(y) + t_3(z) \geq 1; \\ \mathit{false} & \text{if } t_1(x) + t_2(y) + t_3(z) < 1. \end{cases}$$

Now it is not difficult to see that there exists a solution for the CSP if and only if there exists a solution for the SAT problem. Hence, if we have a polynomial algorithm for finding a solution for a CSP, we also have a polynomial algorithm for SAT. According to *Cook's theorem* [?], SATs are NP-complete. So, any problem in NP can be polynomially reduced to the problem whether a CSP has a solution, thus the decision version of a CSP is NP-complete. □

Note that a CSP is an NP-equivalent problem, since every NP-complete problem is self-reducible. For the simplicity, we say that a CSP is NP-complete afterwards.

Although all known algorithms for solving a CSP take super-polynomial time in the worst case, there is a subclass of CSPs that take polynomial expected time. In [?], some classes of binary CSP instances are identified as solvable in polynomial time. One of these classes is the class of instances in which the constraint graph is a tree. Such instances can be solved in $O(nd^2)$, where n is the number of variables and d is the size of the largest domain.

2.3 Solving a CSP

In this section we provide an overview of existing methods and means for solving a CSP. We discuss successively the following topics: possible tasks (2.3.1), search methods (2.3.2), constraint propagation (2.3.3), the order of variable and value selection (2.3.4), dead-end handling (2.3.5), and optimization in CSPs (2.3.6).

2.3.1 Possible tasks

The main task when solving a CSP consists of assigning a value to each variable in such a way that all the constraints are satisfied simultaneously.

Depending on the requirements of an application, the task can be classified into finding:

1. just one solution, with no preference to which one;
2. all solutions; and
3. an optimal solution, where optimality is defined according to prefixed domain conditions as formulated by functions that can be assessed straightforwardly.

We remark that not every CSP is solvable. In this case, the task is at least to prove that no solution exists for the given CSP. Subsequently, one can stop the work, or relax some constraints to find a solution for that case or just attempt to obtain a partial solution that satisfies as many of the constraints as possible.

Whether a CSP is easier or harder to solve is also related to the number of solutions required. This issue will be further discussed in Subsection 2.3.3.

2.3.2 Search methods

In many applications, constructing a search space that covers all possible solutions (i.e., a set of all possible assignments of values to variables) for the problem at hand is a basic method. An appropriate problem representation makes it then possible to move through the search space and to try and find a solution. Frequently the search process is performed in the framework of a graph in which the representations are connected to each other. Solving various kinds of graph problems can be done by an exhaustive graph search which starts from some initial node and ends after the examination of all graph nodes. This is considered as the basic algorithm. Apparently, the exhaustive search of the whole search space is ineffective and only applicable in the case of small search spaces. Within AI, a large number of efficient search methods and strategies have been developed to handle the problems with large search spaces. Some of such methods deal with the problems in which the search spaces can be represented by a special type of graphs, i.e, trees. More discussions on tree search methods and strategies can be found in Bolc and Cytowski's book [**?**].

Search can be very expensive, i.e., it may require a large amount of computation time. For example, searching all truth assignments for a SAT is one of the simplest search procedures. However, it will take a time proportional to 2^n, where n is the number of variables in the problem. Formally, a procedure that terminates in time less than $2^{f(n)}$ for some polynomial $f(n)$ is said to execute in *exponential* time, where n is the size of the problem. So, a *search procedure* is identified as the procedure that requires at least exponential time. A *no-search procedure* is a procedure that only requires polynomial time. This classification of procedures concerns the worst-case time.

From Theorem 1, a polynomial algorithm to solve all CSPs is unlikely to exist. In order to seek for a complete assignment that satisfies all constraints in a CSP, search methods are therefore widely applied.

In fact, a solution of a general CSP can be found by searching systematically through the search space without explicitly using the graph representation. In such a procedure, each possible combination of the variable assignments is systematically generated and tested (this is called Generate and Test 'GT' algorithm) to see whether it satisfies all the constraints. A combination that satisfies all constraints in the CSP is a solution. The number of combinations considered by GT is the size of the Cartesian product of all the variable domains (i.e., $\prod_{i=1}^{n} |D_{v_i}|$, where $|D_{v_i}|$ is the size of D_{v_i} and n is the number of variables in the problem). In other words, GT performs an exhaustive search, and can either find all solutions or prove that no solution exists for the CSP.

Obviously, the GT algorithm is very simple but very inefficient too, because it generates many wrong assignments of values to variables which are rejected in the testing phase. The most common algorithm for performing systematic search involves some notion of backtracking. In this algorithm, all variables involved in the search are selected in some order. The first selected variable creates the *root* node of the search tree (i.e., the start of the searching procedure). Values are assigned to variables one by one. As soon as all the variables relevant to a constraint are assigned, the validity of the constraint is checked. If a partial assignment violates any of the constraints, backtracking is performed to the most recently assigned variable whose domain is still not empty. The search stops if a solution is found, or if all alternative decisions in the *root* of the tree have been tried without solution. In the latter case, the instance of the CSP is said to be infeasible. The backtracking tree search algorithm essentially performs a depth-first search through the search space. It is usually called the *chronological backtracking* algorithm. Figure 2.1 provides an outline of the algorithm.

At each time, variables with assignment are called *past variables*, variables without assignment are called *future variables*; the variable which is being assigned is called the *current variable.* A value $d \in D_v$ is called *inconsistent (consistent)* if no (a) solution exists that includes the assignment of d to v in combination with the partial assignment made so far. When all values of a variable v are inconsistent, it

```
Procedure $Chronological\_Backtracking(V, D, C)$
  $BT(V, \{\}, D, C)$

Procedure $BT(Set\_unassigned, Set\_partial\_assignment, D, C)$
  If $(Set\_unassigned <> \{\})$ do
    select $v \in Set\_unassigned$;
    save $D_v$;
    while $D_v <> \{\}$ do
      select $d \in D_v$;
      $a(v) := d$;
      $D_v - \{d\}$;
      constraint checking;
      If (no inconsistency detected)
        $if(BT(Set\_unassigned - \{v\}, Set\_partial\_assignment + a(v), D, C))$
        return (success);
    recover $D_v$;
  return (failure);
```

Figure 2.1: Chronological backtracking tree search algorithm for CSP

is said that a dead-end is encountered.

Apparently, whenever a partial assignment violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. So, backtracking is better than generate-and-test, however, it has some obvious drawbacks and thus is still inefficient. First, the chronological backtracking may repeat failure due to the same reason. Because it does not identify the real reason of the conflict, the search in different parts of the space may fail for the same reason. Second, chronological backtracking detects the conflict too late. Since it is not able to detect the conflict before the conflict really occurs, i.e., only after assigning the values to the all variables involved in a constraint. If these drawbacks can be overcome, the performance of chronological backtracking can be further improved.

From chronological backtracking tree search algorithm, we can abstract three basic components. They are constraint checking, variable and value selection and dead-end handling. In the successive subsections, we introduce three techniques for overcoming the drawbacks of chronological backtracking. These techniques improve the search efficiency considerably. In 2.3.3 we describe constraint propagation instead of the simple constraint checking for pruning the search space; it makes the search process easier and more efficient. In 2.3.4 we discuss heuristics for variable and value orderings to guide the search process. Finally, in 2.3.5 we describe intelligent backtracking instead of simple chronological backtracking to handle dead-ends adequately.

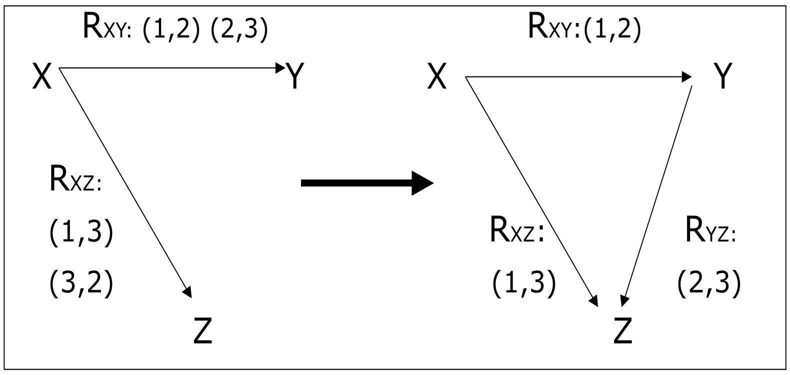


Figure 2.2: A constraint-propagation example

2.3.3 Constraint propagation

Constraints between each pair of variables (X, Y) can be divided into explicit and implicit constraints. Explicit constraints are explicitly recorded in a set of consistent (compatible) value pairs denoted by R_{XY}. However, implicit constraints are not recorded in any place. They can be seen as the side effects caused by the action of explicit constraints. If a value assigned to a variable violates an implicit constraint, the inconsistency will usually not be detected at the time of the assignment, but several steps later, when the set of explicit constraints has acted according to its contexts.

Definition 13 *The deductive processes that make the initially implicit constraints explicit are called* constraint propagation.

A process of constraint propagation is shown in the following example.

Example of constraint propagation Figure 2.2 depicts a constraint graph of a CSP instance (V, D, C), where $V = \{X, Y, Z\}$, and the same domain $\{1, 2, 3\}$ is assumed for all the variables in the CSP. The constraint R_{XY} and R_{XZ} are explicit constraints. After the interaction of the two explicit constraints, the X domain becomes $\{1\}$. This result is propagated by the constraints R_{XY} and R_{XZ}. Then the Y domain becomes $\{2\}$ and the Z domain becomes $\{3\}$. At the end of the process, a new constraint R_{YZ} has arisen that was not explicit in the previous constraint graph.

The constraint checking process in the chronological backtracking algorithm only checks the explicit constraints in the CSP. The implicit constraints are not dealt with. From the example of Figure 2.2, we can see that a simple constraint checking can be improved by constraint propagation. The inconsistent values in the unassigned variable domain that cannot participate in a solution can be detected and filtered out by applying constraint propagation.

In constraint propagation, the most frequently checked forms of consistency are *node* and *arc-consistency.* Node consistency refers to the consistency of a single variable's domain. A variable v's domain D_v is node consistent for a unary constraint c_v, if $c_v(d)$=true holds for all values $d \in D_v$. Node consistency is easily achieved by deleting all values that do not satisfy the unary constraints.

As we focus on binary CSPs, the consistency of a binary constraint should be ensured. In the constraint graph, binary constraints correspond to arcs, therefore this type of consistency is also called arc-consistency. The arc (v_i, v_j) is arc-consistent if for every value $d \in D_{v_i}$ there is some value $d' \in D_{v_j}$ such that $c_{v_i,v_j}(d, d')$=true holds. Note that the concept of arc-consistency is directional, i.e., if an arc (v_i, v_j) is consistent, it does not automatically mean that (v_j, v_i) is consistent too. Clearly, an arc (v_i, v_j) can be made consistent by simply deleting those values from D_{v_i} for which there does not exist a corresponding value in D_{v_j} such that the binary constraint c_{v_i,v_j} is satisfied. Deletions of such values do not eliminate any solution of the original CSP. If all constraints of a binary CSP are arc-consistent (AC), we say that full arc-consistency is achieved. Many algorithms (from AC-3 to AC-7) that achieve arc-consistency have been developed [**?**, **?**, **?**]. Recently, a new algorithm AC2001 was proposed by Bessiere and Regin [**?**]. The time and space complexity of AC2001 are $O(ed^2)$ and $O(ed)$ respectively, where e is the total number of binary constraints and d is the size of the largest domain. For an appropriate overview and some insights into new developments of arc-consistency algorithms, the reader is referred to [**?**, **?**, **?**, **?**].

Arc-consistency in itself is not sufficient to eliminate the need for backtracking. So, a concept called *K-consistency* is introduced. A set of variables is K-consistent if the following requirements hold: choose values of any $K-1$ variables that satisfy all the constraints among these variables and choose any Kth variable. Then there exists a value for this Kth variable that satisfies all the constraints among these K variables. Note that the K-consistency ($K > 2$) is achieved by constraint adaptation on $K-1$ variables rather than using domain reduction. In more detail, if a value tuple that is allowed by the constraints among $K-1$ variables has no consistent value in the Kth variable's domain, this value tuple of $K-1$ variables will be removed.

If the set of variables is *K'-consistent* for all $K' \leq K$, then it is called *strong K-consistent.* The node consistency discussed earlier is equivalent to strong 1-consistency, and arc-consistency is equivalent to strong 2-consistency (arc-consistency is usually assumed to include node-consistency). Algorithms exist for making a CSP strongly K-consistent for $K > 2$, but in practice they are rarely used because of other efficiency issues. If a CSP containing n variables is strongly n-consistent, then a solution to the corresponding CSP can be found without any search (backtrack free). But the worst-case complexity of the algorithm for obtaining n-consistency is also exponential.

The chronological backtracking algorithm performs arc-consistency among already assigned variables for binary CSP, i.e., the algorithm checks the validity of binary constraints on the partial assignment. It cannot detect the conflict before the conflict really occurs. This drawback can be avoided by *forward checking* of the possible conflicts, i.e., by applying arc-consistency between the current variable and the future variables. When a value is assigned to the current variable, any value in the future variable domain which is not arc-consistent with the current assignment is (temporarily) removed. If the domain of a future variable becomes empty (a dead-end is encountered), it is immediately known that the current partial assignment is inconsistent. Forward checking therefore allows that a branch of the search tree that will lead to failure is pruned earlier. Consequently, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables. As a result, checking a new assignment against the past assignments is no longer necessary. Forward checking checks only the constraints between the current variable and the future variables. A more active approach, called *look-ahead*, performs full arc-consistency among the future variables. It also detects the conflicts between future variables and therefore allows that branches of the search tree that lead to failure are pruned earlier than forward checking.

With the implementation of forward checking or look ahead, the constraint propagation not only prunes the search space before the search process starts but also during the search process. More constraint propagations at each node will result in the search tree containing fewer nodes, but the overall cost may be higher, as the computation at each node will be more expensive. For instance, a strong n-consistency CSP would completely eliminate the need for search. However, obtaining the strong n-consistency is usually more expensive than simple backtracking. In some cases even the full look-ahead arc-consistency may be more expensive than simple backtracking combined with forward checking arc-consistency (denoted as FC-AC)[?]. Therefore, FC-AC is still widely used in applications and the implemented constraint propagation is usually incomplete. This means that some but not all the consequences of constraints are deduced. In particular, not all inconsistencies can be detected by constraint propagation.

If a single solution is required, the tighter problems are harder to solve. Since the larger proportion of the search space does not contain a solution, more backtracking is likely to be required. However, when all solutions are required, looser problems becomes harder to solve. Since the problem is loosely constrained, a larger proportion of the search space lead to solutions. In order to find all solutions, the whole search space has to be explored.

As pointed out in [?], the hardest CSPs are neither the most loosely constrained (a solution can quickly be found), nor the most tightly constrained (an inconsistency can be quickly detected), but the intermediate ones. For them it is difficult to establish the consistency or the inconsistency. In principle, the tighter the constraints, the more effectively one can propagate the constraints, which makes problem reduction more effective. Partly because of this, problems with tighter constraints need not be harder to solve than loosely constrained problems. For loose problems, many leaves of the search space represent solutions. A simple chronological backtracking would not require much backtracking before a solution is found. A strategy which combines searching and constraint propagation is likely to spend its efforts unnecessarily in attempting to reduce the problem. From the point of view of the efficiency, it is logical to use different constraint propagation strategies to solve CSPs with different tightness. However, for a large CSP in practice, it is quite difficult to know how many solutions exist before finding all solutions. Consequently, determining the tightness of a large CSP is sometimes much harder than finding one solution. Therefore, the approach that uses a limited form of constraint propagation is widely applied to solve a variety of CSPs [?].

2.3.4 The order of variable and value selection

The order in which the variables are assigned and the values chosen could affect the number of backtracks required in a search. This number is one of the most important factors affecting the efficiency of a search algorithm. So, choosing a good variable and value order to guide the search processes can improve the search efficiency considerably. The simplest strategy for the selection of a next variable and its value is *static ordering*, i.e., the order of the variables and values is predefined before the search starts, and is not changed thereafter. The more sophisticated strategy is *dynamic ordering*, i.e., the choice of a next variable and value at any point depends on the current state of the search.

In the CSP community, several heuristics have been developed and analyzed for variable ordering. One of the powerful heuristics is that the variable with the fewest possible remaining alternatives is selected first for assignment (denoted as FPR). The FPR heuristic provides a substantial improvement over the chronological backtracking for significant classes of problems [?]. Another possible heuristic is to assign those variables first that participate in the highest number of constraints. This heuristic aims at early pruning the unsuccessful branches of the search tree. Sadeh and Fox used a variable ordering (Operation Resource Reliance–ORR) in constraint-based scheduling which is somewhat related to this heuristic [?].

Once a variable is chosen to be assigned, the variable may have several values available. The order in which these values are considered can have substantial impact on the time to find the first solution. One possible heuristic is to prefer those values that maximize the number of options available for future assignments (denoted as MNO). If a variable is selected according to the FPR heuristic, using the MNO value heuristic as performed by the algorithm developed by Kale [?], for instance, is able to solve much larger n-queen problems than without using the MNO value heuristic. Johnston and Minton [?] proposed a *minimal-conflicts* heuristic for value selection. They showed that this heuristic works quite well for *n-queen* problems and for scheduling the Hubble Space telescope. As well as variable ordering, value ordering heuristics are highly problem-specific. For instance, one heuristic may perform quite good for a certain class of problems but poor for another class of problems. Moreover, for a given problem, one variable ordering may perform quite good together with a specific value ordering and poor with another value ordering [?].

In general, a good variable ordering reduces the search efforts by moving the failures to upper levels (closer to the root node) of the search tree. Since a search tree is formed from left to right, A good value ordering moves a solution of the CSP to the left of the search tree so that the solution can be found quickly by the backtracking algorithm. Note that good variable and value orderings are likely to lead to a better run-time performance, but do not change the complexity of the problem.

2.3.5 Dead-end handling

If a partial assignment cannot be extended to a solution satisfying all constraints, we say that a dead-end occurs. The search procedure then has to undo one or more assignments and try alternative assignments. In chronological backtracking, the search undoes the last assignment and tries another value, if it is available, for the selected variable. As a result, chronological backtracking may search exhaustively a sub-tree in which no solution exists. This is because the combination of several earlier assignments causes the algorithm to get stuck in such a sub-tree. The main problem of escaping from a dead-end is to decide which assignments to undo. Besides chronological backtracking, more sophisticated procedures called *Intelligent Backtracking* are developed to escape from a dead-end. They are *dependency-directed backtracking* [?], *backjumping* [?], *graph based backjumping* [?], *learning nogood assignments* [?], etc.[?, ?]. The main idea behind the Intelligent Backtracking is that if backtracking is required, the reasons for the failure are analyzed so that the same mistake can be avoided in the future. Note that a simple Intelligent Backtracking may have less overall complexity than a more complicated Intelligent Backtracking. Because of the complexity of determining the cause of the dead-end, the algorithm of Intelligent Backtracking may take more time in total than even simple backtracking for a variety of problems. Thus, choosing proper dead-end handling is usually combined with the choice of proper constraint propagation, proper variable and value ordering. The optimal combination of these techniques is varied for different problems [?].

In terms of complexity, all the algorithms that integrate constraint propagation, variable and value selection heuristics and dead-end handling techniques have a worst case complexity that is exponential to the number of variables. The reason is that all techniques that attempt to improve the search efficiency do not aim to reach a polynomial complexity, but a better run-time performance. The efficiency improvements will possibly increase the spatial complexity because sometimes the temporal complexities are transferred to the spatial complexities.

2.3.6 Optimization in CSPs

In many practical problems the question whether the solution found reaches the predefined objective is of the utmost importance and therefore optimization is necessary. Normally, the predefined objective is given by an objective function that is regarded as an additional constraint for the CSP. The requirement of the problem is to find an optimal solution that meets the objective function, rather than finding an arbitrary solution. We call these problems *Constraint Optimization Problems* (COPs).

Definition 14 *A COP (V,D,C,g) is defined as a CSP (Definition 2) together with an objective function g which maps every complete assignment to a numerical value. Where (V,D,C) is a CSP, and if S is the set of complete assignments of (V,D,C) then*
$g : S \to \mathbb{R}$.
Given a complete assignment a, we call $g(a)$ the g-value of a.

The task of solving a COP is to find the solution with the optimal (minimal or maximal) g-value. For instance, in job shop scheduling applications, finding just any solution is not good enough. One may like to find the most economical way to allocate the machines to the jobs, minimizing completion time of all jobs, etc.

In general, optimization poses two tasks: finding a feasible solution, and proving that it is optimal. A naive but sometimes costly approach to obtain an optimal solution is finding all solutions so as to compare their g-values. To avoid looking for all solutions, the domain-knowledge sometimes can be used to prune parts of the search space that do not contain solutions better than the best solution ('best' according to the optimization function) found so far. This strategy that incorporates a simple inference into the search processes is realized by *branch and bound* algorithms. In such an algorithm, the bound is initialized to infinity for minimization (minus infinity for maximization) COPs. Then, it searches for solutions in a depth-first manner. It behaves like chronological backtracking, except that before a partial assignment is extended to include a new assignment, the g-value of the current partial assignment is calculated. If the g-value is greater than (less than) the bound, the sub-tree under the current partial assignment is pruned. Whenever a solution is found, the g-value that is less than (greater than) or equal to the bound becomes the new bound. The newly found solution is recorded as (one of) the best solution(s) so far. After all parts of the search space have been searched or pruned, the best solution recorded so far is the solution to the COP.

An optimization problem is called *NP-hard* if the corresponding decision problem is NP-complete. Straightforwardly from Theorem 1, COPs are NP-hard. Usually, the time required to find an optimal solution is much more than that for finding an arbitrary solution. The efficiency is the prominent issue in developing optimization algorithms. Especially in some applications, the environment changes dynamically (e.g., in industrial scheduling, a machine may break down from time to time), an optimal solution at one point may become suboptimal very soon. The decisions derived from the solution could be useless if they come too late. Hence, giving up completeness for speed is crucial in these cases. Consequently, weakening the optimality criteria (i.e., relaxing the additional constraint specified by the objective function) and obtaining a near-optimal solution by approximate algorithms are considered significant in solving such COPs.

For a hard optimization problem, one can crack it by solving a reasonable number of easy problems that are collectively equivalent to the original problem. For instance, when solving a minimization problem, the *upper bound* on the optimal solution value can be quite large at the beginning. When a solution is found, a new upper bound which is lower than the previous one is chosen. So, the upper bound values form a monotonous descending sequence. Each of them corresponds to a new problem which may be easier to solve than the original problem.

When no alternative methods are available, the stochastic search techniques could be used to solve CSP and COP. A stochastic search algorithm moves from one point to another in the search space in a nondeterministic manner, guided by heuristics. The next move in the search space is partly determined by the outcome of the previous move. Generally speaking, a stochastic search is incomplete. Thus, the obtained solution is often near-optimal for a COP.

2.4 Dynamic CSPs

In practice, the set of constraints in a CSP may change over time. For instance, a machine may break down in a job shop. The resource constraint on the operations processed by that machine is thus changed. Such a change may result in a new CSP. The sequence of such CSPs is denoted as a Dynamic CSP (DCSP) [?].

Definition 15 *A dynamic constraint satisfaction problem P is a sequence $P_0, P_1, \cdots, P_i, \cdots$ of static CSPs, each one resulting from a constraint change in the preceding one.*

The constraint change may be a restriction (a new constraint is imposed/added on a subset of variables), a relaxation (a constraint is removed from the CSP) or a combination of restriction and relaxation. We remark that, the other possible changes to a CSP, such as domain modifications, variable additions or removals, can be expressed in terms of constraint additions or removals too.

For any i, if $P_i = (V, C_i, D)$, we have $P_{i+1} = (V, C_{i+1}, D)$ where $C_{i+1} = C_i \pm c$ or $C_{i+1} = C_i - c_1 + c_2$; c, c_1, c_2 being a constraint. Obviously, if a complete assignment $a^\star$ is a solution of $P_i = (V, C_i, D)$ then it is also a solution of $(V, C_i - c, D)$. Upon relaxation, and if there is a previous solution, we may simply keep this solution. Conversely, if a complete assignment $a^\star$ is not a solution of $P_i = (V, C_i, D)$ then it is not a solution of $(V, C_i + c, D)$ either. Consequently, we define the notion of infringed solution.

Definition 16 *A solution of P_i which is not a solution of P_{i+1} is called the infringed solution of P_{i+1}.*

Assume that the constraint change is a restriction or a combination of relaxation and restriction, then a valid solution of one CSP may not be a solution of the next CSP invoked by the new situation. So, we have an infringed solution and a new solution must be generated for that CSP. For solving such CSPs, it is always possible to treat them as independent CSPs and solve them by solution methods introduced in the previous sections. However, any solution obtained in this way might be quite different from a previous one. In practical situations, large differences between successive solutions are often undesirable. So, we must solve, as efficiently as possible, the *solution-maintenance problem* in a DCSP.

Wallace and Freuder [?] recently proposed an algorithm to generate solutions that are expected to remain valid after constraint changes in a DCSP. The algorithm tracks the changes that happen in a DCSP and incorporates this information to guide the search to solutions that are more likely to be maintained. This approach is based on the assumption that the successive changes in a DCSP are temporary and tractable. However, when unexpected events occur, for instance a machine breaks down or an employee becomes ill, the assumptions may no longer be well-founded. As the authors pointed out, in the cases of constraint addition, it is quite complicated to track down the changes directly. Thus, that approach may become too costly to be viable [?].

Verfaillie and Schiex [?] proposed an algorithm by reusing the solution of a previous CSP to find a solution for the new CSP. The solution maintenance was made by two steps. First, the algorithm fixes the assignment of a set of variables in the solution of the previous CSP. Second, it finds a partial assignment for the rest of variables that are consistent with the fixed assignments by the traditional search algorithm. However, for a general DCSP, it is hard to predict which assignment of variables should be fixed in a dynamic environment. Moreover, carrying out the solution maintenance by fixing the assignment of a set of variables is limited in some specific application domains. In other application domains, the objective may be to maintain as many as possible variable assignments equal to the old ones. For instance, for the process of scheduling people working in a hospital where someone becomes ill, it would be disadvantageous if too many changes with respect to the old schedule would occur in night shifts, weekend shifts and corresponding compensation days of all employees. To cope with these application demands, a new algorithm has been developed in our research. It will be discussed in Chapter 4 of this thesis.

Chapter 3

Job Shop Scheduling

Over the last decade a number of powerful methods have been developed for solving *Job Shop Scheduling Problems* (JSSPs) [?]. The success of these methods is mainly due to formulating the JSSP as a CSP and combining the CSP solving methods with the techniques developed in Operation Research (OR), such as *simulated annealing* [?], *edge-finding* [?] etc.[?].

After the precise formulation of a JSSP in Section 3.1, the chapter introduces a useful CSP model of a JSSP in Section 3.2. Then, from Section 3.3 to Section 3.6, an overview of strategies and techniques in constructing a Job Shop Schedule is given, which include constructive algorithms, local-search algorithms, genetic algorithms, and constraint-propagation techniques. Finally, Section 3.7 deals with reactive scheduling issues.

3.1 The Job shop scheduling problem

Scheduling is concerned with the problem of allocating scarce resources to activities over time (i.e., the processes of assigning activities to resources in time). Any manufacturing shop not engaged in mass production of single items will have scheduling problems.

In manufacturing shops a job consists of a series of operations (activities) which are processed by machines (resources) for a certain processing time period. The number of machines available is limited. If a machine can only process one operation at a time, the machine is called a unary capacity machine (resource). If a machine is capable of processing more than one operation at a time, the machine is called a *multiple* capacity machine (resource). In order to distinguish machines from other resources in the job shop scheduling problem, we use the notation of machine to denote process resource. Except for explicit explanation, the machines mentioned are unary capacity machines. For description convenience and consistency, we use operations instead of activities.

Usually, the operations of a job cannot be processed in arbitrary orders but are subject to a prescribed processing order. From this point of view, manufacturing shops can be categorized in three types: (1) if all jobs pass the machines in an identical order, it is called a flow shop; (2) if jobs pass the machines in predefined different orders, it is called a job shop; (3) if the operations of the jobs can be processed in arbitrary orders, it is called an open shop. Since a job shop contains a variety of constraints on the order of the operations, the problem that determines the start times of the operations in a job shop is more complicated than in a flow shop and an open shop. As a result job shop scheduling is considered to be a good representation of the general domain and has earned a reputation for being notoriously difficult to solve.

The task of job shop scheduling is to determine the start times of the jobs' operations such that the prescribed orders (if they exist) are not violated and the processing times of identical machines do not overlap in time. The resulting time table is called a schedule. In practice, each scheduling pursues at least one economic objective. Typical objectives are: (1) the reduction of the *make-span* of an entire production program (i.e., the maximum completion time of all operations) [?], (2) the minimization of mean job *tardiness* (i.e., how much time the operations finish after their due date) [?], (3) the maximization of machine load and (4) some weighted average of many similar criteria.

In order to handle JSSPs properly, we adopt a formal definition of JSSP as given by Nuijten et al. [?], with some small adaptations to our notation.

Definition 17 *An instance of a standard JSSP is a tuple $(\mathcal{J}, \Omega, \mathcal{M}, H, \prec, J, M, p)$, where $\mathcal{J}$ is a set of jobs, Ω is a set of operations, $\mathcal{M}$ is a set of machines and $H \in \mathbb{N}$ is an scheduling horizon. A function $J : \Omega \rightarrow \mathcal{J}$ gives each operation the job to which it belongs, a function $M : \Omega \rightarrow \mathcal{M}$ gives each operation the machine on which it must be processed, and function $p : \Omega \rightarrow \mathbb{N}$ gives the processing time of each operation. A binary relation $\prec$ is used to decompose Ω into chains, such that every chain corresponds to a job. A schedule is a function $st : \Omega \rightarrow \mathbb{N}$ which denotes the nonnegative start times of the operations. The problem is to find a schedule st such that for all $o, o' \in \Omega$:*

1. *$st(o) + p(o) \leq H$,*
2. *$st(o) + p(o) \leq st(o')$, if $o \prec o'$ and,*
3. *$st(o) + p(o) \leq st(o')$ or $st(o') + p(o') \leq st(o)$, if $M(o) = M(o')$ and $o \neq o'$.*

   where $o \prec o'$ means that the operation o is before the operation o', $M(o)$ denotes the machine on which the operation o is processed.

Building a system to solve job shop scheduling problems is not a trivial task. The scheduler or the system developer has to be aware of three issues before starting work.

- The complexity of the job shop scheduling problem. Most job shop scheduling problems are known to be NP-hard [**?**]. In practice, this means that one must design robust approximate algorithms to generate an appropriate (possibly optimal but often sub-optimal) solution in a bounded amount of time. Depending on different applications, the response time required for constructing a schedule may vary from a few microseconds to a few days.

- The specification of the problems. Different processing environments induce different scheduling constraints that more or less contribute to the complexity of the problem, some of which may be very specific to the problem under consideration. The size of a scheduling problem may vary from a few dozens of operations to thousands of operations. Thus, the algorithms that work well on the small problems may not be applicable to bigger problems.

- The integration with the overall processing system. A scheduling system must get its data from the information system globally in use in the job shop, and must return its results (i.e., the constructed schedule) for shop-floor execution.

3.2 Constraint-based job shop scheduling

Over the last decade, CSP methodologies have successfully been used for handling JSSPs in the AI community [**?**, **?**]. The constraint representation of a JSSP is able to express the problem knowledge at a deep level and may form the basis for search guidance in the solution search processes.

3.2.1 CSP model of a JSSP

Based on the Definition 17, we give the following CSP definition of a JSSP.

Definition 18 *A JSSP* $(\mathcal{J}, \Omega, \mathcal{M}, H, \prec, J, M, p)$ *is a CSP* (Ω, D, C) *which involves a set* $\Omega = \{o_1, o_2, \cdots, o_n\}$ *of* n *operations, a set of operation start time domains* $D = \{D(o_1), D(o_2), \cdots, D(o_n)\}$ *and a set* $C = \{C_1, C_2, C_3\}$ *of constraints. Where each* C_l $(1 \leq l \leq 3)$ *denotes a set of constraints that are defined on the subset of* Ω*:*
$C_1 : st(o_i) + p(o_i) \leq H$*;*
$C_2 : st(o_i) + p(o_i) \leq st(o_j)$*, if* $(J(o_i) = J(o_j) \wedge o_i \prec o_j)$*;*
$C_3 : st(o_i) + p(o_i) \leq st(o_j)$ *or* $st(o_j) + p(o_j) \leq st(o_i)$*, if* $M(o_i) = M(o_j)$*;*
where $o_i, o_j \in \Omega$ $(1 \leq i \leq n, 1 \leq j \leq n)$*.*

From the point of view of a constraint satisfaction problem, two main types of constraint are considered in a JSSP. (a) *Temporal constraints* (i.e., precedence constraints, denoted by C_2) between two operations in the same job specify that if operation o_i is before o_j in the total order ($o_i \prec o_j$), then o_i must execute before o_j. (b) *Capacity constraints* (machine constraints, denoted by C_3) specify that no

$st(o)$:	A variable representing the start time of o;
$D(o)$:	The discrete domain of possible values for $st(o)$;
$J(o)$:	The job to which operation o belongs;
$M(o)$:	The machine on which operation o is processed;
$p(o)$:	Processing time of o;
$P(\Omega)$:	The sum of the processing times of all operation in Ω;
$M(\Omega)$:	The machine on which all operations $o \in \Omega$ are processed;
$J(\Omega)$:	The job to which all operations $o \in \Omega$ belong.

Figure 3.1: Notations

two operations requiring the same machine can be executed simultaneously. In a real job-shop floor, jobs have release dates (the time after which the operation in the job can be executed) and due dates (the time by which the last operation in the job must be finished). These requirements are often added to the constraint set to determine each operation's earliest start time and latest finish time. The job shop scheduling problem thus becomes to determine whether there is an assignment of a start time to each operation such that all constraints defined in Definition 18 are satisfied. This approach is called *constraint-based scheduling*.

Many scheduling problems are not simply CSPs but COPs (Constraints Optimization Problems). To handle these problems, the optimization functions are added as new constraints. Some relatively simple optimization functions have been studied in the literature such as the minimization of make-span [**?**], minimization of the average (or maximum) tardiness of operations, or some combination of other attributes (for example, minimize work-in-process combined with tardiness [**?**]).

For the sake of consistently describing JSSPs, we use the notations given in Figure 3.1 throughout the context of this thesis (where $o \in \Omega$).

3.2.2 Constraint-directed search for solving JSSPs

The fundamental technique used to solve a JSSP in constraint-based scheduling is the *constraint-directed search*. This search technique utilizes the constraint representations not only to model the problem knowledge but also to guide search to a solution. The constraint-directed search algorithms can be described by the framework in Figure 3.2 (referred to [**?**]).

Since a variable has been introduced for each operation, we speak of operation selection instead of variable selection. Similarly, we often use start-time selection instead of value selection.

Like the chronological-backtracking algorithm, the Framework of constraint-directed search contains three basic components. They are constraint propagation,

```
While not solved and not infeasible do
    constraint propagation
    if a dead-end is detected then
      dead-end handling
    else
      select variable
      select value for variable
    endif
endwhile
```

Figure 3.2: Framework of constraint-directed search.

operation and start-time selection, and dead-end handling. More specifically, each time an operation is assigned a start time, inconsistent start times of the unassigned operations are removed by the consistency checking algorithm. The new start-time domain of an unassigned operation is called its current start-time domain. If a partial assignment obtained cannot be extended to a solution satisfying all constraints, we say that a dead-end occurs. The purpose of the next operation and its start time selection is to avoid getting trapped in a dead-end. They are done by operation and start time selection heuristics. When a real dead-end occurs, the procedure called dead-end handling undoes one or more previous assignments and tries alternative assignments. The search stops if a solution is found, or if all alternatives have been tried without success. In the later case, the instance is said to be infeasible.

Based on the notations in Subsection 3.2.1, we introduce the following definitions. For each operation $o \in \Omega$, $D(o)$ is its current start-time domain.

- $est(o) = min\{t \mid t \in D(o)\}$ is the earliest possible start time of o;
- $eft(o) = min\{t + p(o) \mid t \in D(o)\}$ is the earliest possible finish time of o;
- $lst(o) = max\{t \mid t \in D(o)\}$ is the latest possible start time of o;
- $lft(o) = max\{t + p(o) \mid t \in D(o)\}$ is the latest possible finish time of o;
- $EST(\Omega) = min\{est(o) \mid o \in \Omega\}$ is the minimal earliest possible start time of all operations in Ω;
- $EFT(\Omega) = min\{eft(o) + p(o) \mid o \in \Omega\}$ is the minimal earliest possible finish time of all operations in Ω;
- $LST(\Omega) = max\{lst(o) \mid o \in \Omega\}$ is the maximum latest possible start time of all operations in Ω;
- $LFT(\Omega) = max\{lft(o) + p(o) \mid o \in \Omega\}$ is the maximum latest possible finish time of all operations in Ω;

In the following sections, we first introduce a number of constructive algorithms that generate schedules in constraint-based scheduling approach. Then, we briefly introduce other approaches. We end the chapter with a discussion on reactive scheduling.

3.3 Constructive algorithms

The constructive approach starts to create a job-shop schedule from scratch, then works on a consistent partial solution (that is, a fixed subset of operations successfully assigned) and attempts to extend it by assigning a start time to a currently unassigned operation. Notable progress has been made in this approach in the last decade [**?**, **?**, **?**, **?**, **?**].

3.3.1 An optimization algorithm

Baptiste et al. [**?**] presented an optimization algorithm to find a schedule with the minimal make-span. The algorithm starts with the propagation of temporal constraints to determine whether a set of temporal constraints is consistent. It also determines earliest and latest start and finish times for operations that are globally consistent with all the temporal constraints. As a result, solutions to the JSSP can be obtained by sequencing all the operations that require a common machine. Given an upper bound for the make-span, the following constructive algorithm either finds a solution to the JSSP or proves that no solution exists:

1. Select a machine among the machines required by unordered operations.
2. Select the operation to execute first (or last) among the unordered operations that require the chosen machine. Post and propagate the corresponding precedence constraints. If an inconsistency is detected, a backtrack occurs. Keep the other operations as alternatives to be tried upon backtracking.
3. Iterate step 2 until all the operations that require the chosen machine are ordered.
4. Iterate step 1 and 3 until all the operations that require a common machine are ordered.

Effective heuristics are used to select the machine and the operation to execute either first or last among the unordered operations that require the machine.

To find a schedule with the minimal make-span, the algorithm proceeds by branch-and-bound search (or binary search on the make-span interval). As long as the above search algorithm succeeds in generating a solution to the problem, a new constraint stating that the value of the criterion to minimize must be strictly smaller than the value of the best solution found so far is added to minimize make-span. When the search algorithm fails, i.e., reports that there is no better solution, it is known that the best solution found so far is optimal.

3.3.2 Approximation algorithms

Although the results obtained by the optimization algorithms are quite good, a significant portion of the CPU time is spent to find a solution relatively far from the optimal value. The reason is as follows: when the upper bound make-span of a scheduling is much higher than the optimal make-span, there are many solutions, making the algorithm easy to find one. Furthermore, when the upper bound is very close to the optimal make-span, constraint propagation becomes very effective in pruning the search space. Although it is not easy to find a solution, constraint propagation provides reliable guidance. However, for intermediate values of the upper bound, it is fairly difficult to find a solution and constraint propagation does not provide much guidance. This increases the probability of taking a wrong decision and as a chronological-backtracking (systematic) search strategy is used, it may take long to recover from such a mistake.

An alternative to an optimization algorithm is an approximation algorithm. Several types of approximation algorithms are available for JSSPs. Below we discuss three of them.

(1) Applegate and Cook [**?**] use the shuffle procedure to improve solutions to the JSSP. The basic ideas is to fix a number of ordering decisions, based on the best solution found so far, and search for an optimal schedule among those that respect those decisions. This procedure gives very good results, especially when effective constraint propagation techniques are used. The reason is that the propagation of the imposed decisions results in a drastic reduction of the search space and thus enables a fast resolution of the remaining sub-problems.

(2) Nuijten [**?**] points out that every schedule can be transformed into a left-justified schedule in which the operations are scheduled as early as possible while preserving the precedence constraints and the machine orderings. So, he uses the simpler and less expensive heuristics in which the operations and start times that together construct left-justified schedules are selected. In more detail, an operation is selected by determining the minimal earliest finish time of any unscheduled operations and then randomly selecting one operation that can be started before this finish time. The start-time selection consists of selecting the earliest possible start time of this operation. As a result the number of possible decisions in each node of the search tree is drastically reduced. Moreover, Nuijten uses a more powerful propagator — edge-finding to derive more precise start time domains for the unassigned operations. This further reduces the number of possible decisions in each search state.

To escape from dead-ends, a randomized procedure is applied. First, the chronological backtracking is used in order to solve the instance. If this does not lead to a solution after a reasonable number of backtracks, the search is stopped and then completely restarted. The problem then is to direct the search along a path different from the ones followed previously. By restarting the search with a randomized selection of a next operation, the probability of following the same search path more than once is very small since the number of possible paths is usually very large.

Furthermore, by backtracking, if it is derived that operation o cannot be scheduled on time t, it is implied that it cannot be started on any time in the interval $[t+1, \min\{eft(o') \mid o' \in \Omega_{M_{(o)}} - \{o\}\})$, where $\Omega_{M_{(o)}}$ represents the set of operations which require the same machine as o does.

(3) An algorithm presented by Baptiste et al. in [?] combines the two ideas above. At each step of the algorithm, a number of ordering decisions is kept. Then as in (1), the algorithm proceeds to search for an optimal schedule among those that respect the decisions. However, the decisions that are kept are randomly selected and the search is stopped after a given number of backtracks.

The algorithm can be refined by introducing different operation selection heuristics. Each of these heuristics is tried in turn. The procedure terminates when N iterations of each of the heuristics have failed to produce a solution better than the best available solution. Finally, the approximation and optimization algorithms can be combined as follows: when the approximation algorithm terminates, the optimization algorithm is launched.

3.3.3 The ORR-FSS algorithm

The ORR-FSS algorithm was proposed by Sadeh and Fox in [?, ?]. In the algorithm, each iteration works on a consistent partial assignment and attempts to extend it by assigning a start time to a currently unassigned operation. ORR (Operation Resource Reliance) heuristically identifies the most critical operation by finding the operation that relies most upon the machine and the time for which there is the most contention (i.e., the highest number of operations that are competing to execute at a particular time on a machine). FSS (Filtered Survivable Schedules) then rates the quality of the possible start times of the critical operation. The start time with the highest quality (which is expected to be compatible with the largest number of survivable job schedules) is assigned. Chronological backtracking, temporal-constraint propagations and capacity-constraint propagations are used in the algorithm. The algorithm terminates if all operations are assigned a consistent start time, or a user-specified bound on the number of assignments is reached.

Although ORR-FSS uses sophisticated heuristics for the operation and the start-time selection, experimental results show that the performance of ORR-FSS is poor in comparison with Nuijten's algorithm, even if ORR-FSS is augmented with the propagators used in Nuijten's algorithm [?].

3.4 Local-search algorithms

Local-search techniques are useful tools for solving *discrete optimization* problems. All non-preemptive scheduling problems are discrete optimization problems. A discrete optimization problem can be described by the following definition.

Definition 19 *For a given finite set S and a given cost function $c : S \rightarrow \mathbb{R}$, one has to find a solution $s^\star \in S$ such that: $\forall s \in S \;\; c(s^\star) \leq c(s)$.*

Local search is an iterative procedure which moves from one solution in S to another as long as necessary. In order to search systematically through S, the possible moves from a solution s ($s \in S$) to the next solution is restricted by the *neighborhood* $N(s)$, where $N(s)$ describes the subset of solutions which can be reached in one step by moving from s. By selecting a solution $s' \in N(s)$ based on the values $c(s)$ and $c(s')$, a starting solution of the next iteration is chosen. According to different criteria used for the choice of the starting solution of the next iteration, different types of local search techniques can be utilized.

The simplest choice is to take the solution with the smallest value of the cost function. This choice leads to the well-known *iterative improvement* method. In general, a final solution $s^\star$ obtained by iterative improvement is only a local minimum with respect to the neighborhood N (i.e., a solution such that no neighbor is better than this solution) and may differ considerably from the global minimum. A method that seeks to avoid being trapped in a local minimum is *simulated annealing* [**?**]. In this method, s' is chosen randomly from $N(s)$. In the i^{th} step s' is accepted with probability $min\{1, exp(-\frac{c(s')-c(s)}{c_i})\}$, where c_i is a sequence of positive control parameters with $\lim_{i\rightarrow\infty} c_i = 0$.

A variant of simulated annealing is the *threshold acceptance* method. It differs from simulated annealing only by the acceptance rule for the randomly generated solution $s' \in N(s)$. s' is accepted if the difference $c(s') - c(s)$ is smaller than some non-negative threshold T, where T is a positive control parameter which is gradually reduced.

Simulated annealing and threshold acceptance still have the disadvantage that it is possible to get back to solutions already visited. A simple way to avoid such problematic returns is to store all visited solutions in a list called *tabu list* and to accept only solutions which are not in the list. This local search technique is called *tabu search*. Further discussions about tabu search can be found in [**?**, **?**].

Zweben et al. presented a local search algorithm for scheduling problems in [**?**]. The algorithm, called GERRY, begins with a complete assignment of start times that fails to satisfy some set of constraints and then iteratively modifies or *repairs* the start times of some operations to improve the quality of the schedule. The quality of a schedule is measured by a cost function which is a weighted sum of constraint violations.

Gerry preserves temporal (precedence) constraints in all its problem solving activities. Therefore, when applied to job shop, the only constraints that Gerry repairs are the machine capacity constraints. If K is the set of operations contributing to the violation, Gerry tries to move each operation in K to the previous and next times at which the machine is available. Each candidate move is evaluated by a linear combination of a number of factors, including the extent to which the processing time of the operation matches the amount of the over-allocation, the number of operations temporarily dependent on the operation, and the distance from the current start time of the operation to the new start time. The repair then converts each evaluated value into a probability, and a operation is selected based on the probabilities.

At the end of each iteration, the overall cost of the schedule is calculated. If the cost is less than that of the previous schedule, the new schedule becomes the current schedule for the next iteration. If the new schedule is better than any previous solution, it is cached as the schedule "so far best". Even if the schedule has a higher cost than the previous schedule, it may be accepted by some probability based on the simulated annealing technique in order to escape local minima and cycles. As the search progresses, a higher cost schedule is increasingly less likely to be accepted. The algorithm terminates when the quality of the obtained schedule is found to be satisfactory or when a designated time bound is reached.

3.5 Genetic algorithms

Genetic (Evolutionary) algorithms are probabilistic algorithms. An initial population of likely problem solutions (rather than a single solution) is usually created by some randomized means. The algorithm is assumed to evolve towards better solution versions. New solutions are generated with the use of genetic operators patterned according to the reproductive processes observed in nature. Each element of a current solution space (population) is called a chromosome, and its components (there is a number of components) are called genes. Because of the similarity in action, genetic operators have names originating from genetics: cross-over, mutation, and inversion. However, in a variety of applications different representations can be used which may lead to the use of specially adapted genetic operators.

After a certain number of generations, when the successive populations (offspring) are no longer getting better (in the sense of a fitness function), the best chromosome represents the optimal solution. In practice, the algorithm terminates after a number of iterations (having taken into account storage and time constraints).

Many researchers have worked on solving JSSPs by using the genetic algorithm template [**?**, **?**, **?**, **?**, **?**, **?**]. In recent years, hybrid genetic (evolutionary) search scheduling algorithms (ESSA) are developed to overcome disadvantages of the stand-alone genetic algorithm [**?**, **?**]. They combine a local search algorithm to update the offsprings, and try to optimize locally the created offspring as much as possible before it is merged in the population. However, the new method is in principle not different from the basic genetic algorithm template. The major differences are (1) the representation (encoding) of a schedule (chromosome), (2) the specific operators used to produce offsprings (e.g., crossover and mutation), (3) the fitness function used to evaluate the chromosome. Although Hybrid ESSA performs quite well to find optimal schedules for some benchmark JSSPs [**?**], many of them cannot handle an infeasible complete assignment for a JSSP. Moreover, the computation time needed (i.e., efficiency) is not addressed in full and not even in its essence in the genetic algorithm literature.

3.6 Constraint propagation techniques

In recent years, many powerful propagation techniques have been put in use for constraint-based scheduling. For instance, variations of edge-finding [**?**, **?**, **?**] have been integrated in many constraint-directed search algorithms. Although, it has long been known that the search can drastically be reduced by enforcing various degrees of consistency implemented by constraint propagation algorithms, the effort to achieve such high degrees of consistency appeared to be at least as expensive as the traditional algorithms. Therefore, the goal for the research with emphasis on propagation is to find a trade-off between complexity and the resultant easing of the search effort. We briefly introduce four efficient propagator techniques in the following subsections, namely the time-table propagator, the disjunctive constraint propagator, the edge-finding technique, and the energy-based reasoning technique.

3.6.1 Time-table propagator

The *time-table* propagator defines a data structure called timetables to maintain information about machine utilization and machine availability over time. The data structure can be discrete arrays or sequential lists which contain constrained operations. The machine constraints are propagated in two directions: (1) from the machines to the operations, in order to update operation time-bounds according to the availability of the machines; (2) from the operations to the machines, in order to update machine utilization and availability according to the time-bounds of operations. For example, when $lst(o) < eft(o)$, it is sure that the operation o will use the machine within the interval $[lst(o), eft(o))$. Over this period, the corresponding machine is no longer available for other operations.

3.6.2 Disjunctive constraint propagator

The *disjunctive* constraint propagator applies to unary machines. It consists of posting a generic disjunctive constraint to ensure that the time intervals over which two operations require a unary machine cannot overlap in time. Such disjunctive constraint propagation is more time-consuming but often results in more precise time bounds than the propagation of the corresponding time-table constraints. The most basic disjunctive constraint states that two operations o and o' which require the same unary machine M cannot overlap in time: either o precedes o' or o' precedes o. This can be stated as follows:

$[end(o) \leq start(o')]$ or $[end(o') \leq start(o)]$

In this formula, $start(o), end(o), start(o'), end(o')$ denote constrained variables. Constraint propagation consists in reducing the set of possible values for these variables: whenever the minimal possible value of $end(o)$ exceeds the maximal possible value of $start(o')$ (i.e., $eft(o) > lst(o')$), o cannot precede o'; hence o' must precede o; the time-bounds of o and o' can consequently be updated with respect to the new temporal constraint $[end(o') \leq start(o)]$. Similarly, when $eft(o') > lst(o)$, o' cannot precede o. When neither of the two operations can precede the other, a contradiction is detected.

The disjunctive formulation above does not necessarily imply the explicit creation of a disjunctive constraint for each pair of operations. To create a unique global constraint for the overall set $\{o_1, \ldots, o_n\}$ of operations that require a given unary machine M is equivalent to applying the process described above to the $n(n-1)/2$ pairs of operations $\{o_i, o_j\}(1 \leq i, j \leq n)$. In fact, it is an arc-consistency enforcement of disjunctive constraints over the start-time domains of all operations.

3.6.3 Edge finding

The problem of eliminating all the impossible start and end times of operations submitted to machine constraints is NP-hard. Even when only one unary machine is considered, the problem of determining whether there exists a schedule satisfying the given time-bounds for each operation is NP-hard [?]. Two types of methods that provide more precise time-bounds have been developed. The first type of method, called edge finding, consists in determining whether an operation o must, can, or cannot be the first or the last to execute among a set Ω of operations that require the same machine. The second type of method, called energy-based reasoning, consists in comparing the amount of machine energy required over a time interval [t1 t2) to the amount of energy that is available over the same interval. This form of propagation will be discussed in the next subsection.

One of the most successful algorithms for updating time-bounds of operations submitted to unary machine constraints was proposed by Carlier and Pinson [?] ; it is called edge-finding technique. The main principle of this algorithm is to compare the temporal characteristics of an operation o to those of a set of operations Ω that require the same machine. Assume that a set of operations Ω on one machine is selected. For each operation $o \in \Omega$, o is constrained to execute first (or last) among the operations in Ω. Then the edge-finding rule presented below deduces that some operations must, can, or cannot, execute first (or last) in Ω.

$LFT(\Omega \cup \{o\}) - EST(\Omega) < p(o) + P(\Omega) \Rightarrow o \prec \Omega$

$LFT(\Omega) - EST(\Omega \cup \{o\}) < p(o) + P(\Omega) \Rightarrow o \succ \Omega$

Where $o \prec \Omega(o \succ \Omega)$ denotes that o is before (after) all operations in Ω.

Consequently, new time-bounds can be deduced. When o is before all operations in Ω , the end time of o is at most $LFT(\Omega) - P(\Omega)$; When o is after all operations in Ω, the start time of o is at least $EST(\Omega) + P(\Omega)$.

If n operations require the same machine, there are potentially $O(n \times 2^n)$ pairs $\{o, \Omega\}$ to be considered. The computation complexity is exponential. However, Carlier and Pinson [?] present an algorithm that performs all of the possible time-bound adjustments in $O(n^2)$. It consists of a "primal" algorithm to update earliest start times and a "dual" algorithm to update latest finish times. The "primal" algorithm consist of computing "Jackson's preemptive schedule" (JPS)[?] for the machine under consideration. A variant of this algorithm, presented by Nuijten in [?] has the same complexity but does not require the computation of Jackson's preemptive schedule. Carlier and Pinson [?] present a variant running in $O(n \times log(n))$, too.

Caseau and Laburther [?] present two sets of rules for the edge-finding technique based on the concept of "task intervals". There are at most $O(n^2)$ task intervals to consider for a machine on which n operations are scheduled. Therefore, a computational time in $O(n^3)$ is, in the worst case, necessary to apply the rules to all the tasks intervals and all the operations of a given machine.

While integrating the edge-finding algorithm in a generic constraint propagation framework to deduce more precise start-time bounds for operations, the efficiency issues have to be considered too. When both temporal constraints and capacity (machine) constraints apply, it is intuitively more efficient to deduce all the consequences of temporal constraints prior to the application of the edge-finding technique. The main reason is that the propagation of each temporal constraint has a very low cost in comparison to an application of edge finding. The edge finding is consequently delayed until the other constraints have been propagated.

3.6.4 Energy-based reasoning

Energy-based reasoning (EBR) consists in comparing the amount of machine energy required over a time interval [t1 t2) to the amount of energy that is available over the same interval. Erschler et al. [?] analyzes the effect of time and machine constraints on the admissibility of schedules. They studied how operation characteristics and machine constraints can induce new constraints which allow to restart the propagation. The definition of required energy consumption of an operation o over an interval $[t_1\ t_2)$ is the smallest amount of time during which o will be executed on the interval:

$$W_o^{[t_1\ t_2)} = min(p(o), t2 - t1, est(o) + p(o) - t1, t2 - lft(o) + p(o))$$

$W_o^{[t_1\ t_2)}$ is null if the previous quantity is negative. Two deduction rules apply.

- The first rule is based on the idea of picking two operations o and o' and trying to find a contradiction when sequencing o before o'. To achieve this goal, the required consumption of all the operations o'' over the interval $[est(o)\ lft(o'))$ is computed and compared to the provided amount of machine energy during the same interval.

  $lft(o') - est(o) < \sum_{o'' \notin \{o\, o'\}} W_{o''}^{[est(o)\ lft(o'))} + p(o) + p(o') \Rightarrow o' \prec o$

- The second rule directly updates time-bounds. An operation o and an integer X in the interval $[est(o)\ lft(o))$ are chosen. Then checking on the interval $[est(o)\ X)$ whether o can or cannot start at its earliest possible start time. If it cannot start, then $est(o)$ will be increased.

  Let be $\sum W = \sum_{o'' \neq o} W_{o''}^{[est(o)\ X)} + min(p(o), X - est(o))$.

  The deduction rule is:
  $X - est(o) < \sum W \Rightarrow est(o) := est(o) + \sum_{o'' \neq o} W_{o''}^{[est(o)\ X)}$

There are many values of X for which this rule could be used. Erschler et al. [?] suggested that the appropriate values of X are the earliest and latest start and finish times of operations (which seems reasonable but is yet to be proven).

The complexity of *energy-based reasoning* is $O(n^3)$. Baptiste and Le Pape [?] pointed out that EBR is quite efficient for solving very hard job-shop scheduling problems. But for most benchmarks of JSSPs, as we found in experiments, adding EBR to a scheduling algorithm which has already enforced arc-consistency and edge-finding does not outperform the algorithm without EBR.

When the amount of propagation is extended, the search space is more drastically pruned. Due to the management of the additional data structures used to implement these techniques, more CPU time and memory consumption may be required, but the number of problem-solving steps decreases.

3.7 Reactive scheduling

Traditionally, scheduling focuses on optimization of performance under idealized assumptions of environmental stability and solution executability. However, in industrial environments, some schedule requirements cannot be fixed in the pre-establishing phase. They may become manifest only as the time of execution comes closer or when some unexpected events happen. These dynamically evolving and changing requirements may continually force reconsideration and revision of a pre-established schedule. For instance, while a pre-established schedule is executing, a machine may break down, an operator may be absent, the supplier delivery may change (potentially resulting in a lack of materials). Whatever the reason, quick reactions are required to handle these unforeseen events in a job shop. The scheduling systems which cope with dynamic changes in the execution environment are called reactive scheduling systems [?]. Comparatively, generating a schedule before its execution (pre-establishing a schedule) is called predictive scheduling [?].

Over the past decade, many reactive scheduling systems have been developed. Most of these systems emphasized the importance of keeping the continuity of execution and the real-time response in order to reduce disruption in the existing schedule. This means that the original schedule should be modified to the minimum extent possible. Moreover, the time required to get or validate a new schedule should not exceed the time window in which the schedule must be applicable. For more elaborate discussions of reactive scheduling we refer to [?, ?, ?, ?, ?].

In general, there are three reactive strategies that a reactive scheduling system can have to the occurrence of environmental changes. First, do not attempt any reaction. It results in not taking advantage of any opportunities or in incurring execution delays entailed by deleterious events (e.g., partial or total loss of machine capacity). This strategy is obviously suboptimal. The second reactive strategy could be to throw away the rest of the schedule and reschedule from the point of the occurrence of the environmental changes. The rescheduling tools might be the same ones as those that created the predictive schedule. Such strategy may efficiently produce high-quality schedules but may increase schedule disruption. The third reactive strategy could be incremental revision/repair of the existing but flawed schedule. The incremental repair process presented in [?] is attractive not only for incrementally improving a sub-optimal and cheap schedule to meet optimization objectives but also in response to the environmental changes during the schedule execution.

The majority of reactive scheduling systems that appeared in the literature concentrates on the third strategy. OPIS [?] and CABINS [?] are typical such systems. In OPIS, constraint analysis is used to recognize the conflict in the current schedule to identify important modification goals, and to estimate the possibilities for efficient and non-disruptive schedule modification. This information, in turn, provides a basis for selecting among a set of alternative modification actions that range from general heuristic search procedures to more specialized procedures. The general heuristic search procedures in OPIS orient toward generating and revising sets of decisions associated with a specific process or machine. The specialized procedures are used for sliding schedule components forward or backward in time and performing pairwise machine assignment exchanges.

CABINS uses case-based reasoning (learning) to bias the search procedures for incrementally revising a complete but flawed schedule. It is based on the idea that a reactive scheduler usually encounters many times quite similar situations. Thus, the reactive scheduling can benefit from a learning mechanism which can capture and reuse of the previous reaction knowledge. A set of repair actions is presented in CABINS. Each of them operates with respect to a particular local view of the problem and offers selective advantages for improving schedule quality. The goals of this approach are (1) to arrive at a schedule that does not violate any constraint,

(2) to optimize the modified schedule according to the user's preference and (3) to minimize the schedule disruption.

In the *incremental repair* strategy, the problem is additionally complicated by the *ripple* effects that spread conflicts to other parts of the schedule as actions are applied and specific revisions are made. The constraint propagation is then used to propagate the ripple effects of a schedule repair action. To avoid unnecessary computation and to limit the amount of ripple effects, OPIS, by analyzing conflict features in the current schedule, selects a focal-point machine and operation(s) to take proper repair actions. In contrast, CABINS adopts a different approach. The repair action of CABINS is job oriented, where the focal job under the current repair consideration is randomly identified. Repair is performed by one operation at a time. Operations in the focal job are repaired in a forward fashion starting from the earliest operation of that job that has a specific feature.

Recently, Sakkout [**?**, **?**] proposed a *Probe Backtrack Search* to reconfigure as minimal as possible schedules in response to a changing environment. They use a very close integration of constraint-directed search with linear-programming (LP) optimization. The approach conducts a search in two interleaved phases, i.e., a resource feasibility phase and a temporal optimization phase. In the resource feasibility phase, the algorithms use a unimodular probing backtrack search to relieve resource contention by ordering temporal variables. In the temporal optimization phase, the algorithms use cheap linear solving procedures to find values for the temporal variables that are optimal, subject to the orderings decided in the resource phase. At each search step, the algorithm implements a repair step (as follows: it uses LP to generate an "optimal solution" on a subset of constraints, it selects a still violated constraint and it rules out the current "optimal solution" by imposing an "easy" constraint that maintain the unimodularity property) until no violated constraint exists. It apparently adopted an incremental repair strategy.

Some other research focused on generating *robust schedules* (*solutions*) [**?**, **?**] that are likely to remain valid after minor changes to the problem. Nevertheless, major changes (for instance, a machine breaks down) usually make the modifications inevitable to even very robust schedules.

Both rescheduling and incremental repair strategies may utilize *model modification* to the occurrence of unexpected events in a job shop. Model modification reformulates the problem by changing model parameters, such as excluding a number of operations which have already been executed to be rescheduled (repaired), relaxing global constraints by postponing due dates or buying a new machine etc. Model modifications facilitate the solution of the problem, since they amount to global constraint relaxation. However, model modifications are costly to implement in practice (e.g., buy new equipment, pay a fine for the production delay, subcontract jobs to outside contractors). Thus model modifications should be made to the minimum extent possible.

No experimental evidence has been provided so far in favor of incremental schedule repair as opposed to rescheduling. Like for predictive scheduling problems, no polynomial-time solution algorithm is known for reactive scheduling problems. Computation times are usually highly variable and unpredictable [**?**]. In order to bridge the gap between capabilities and requirements, a reactive scheduling system is considered as real-time response only in the simplest sense in which the program runs fast enough to cope with the job shop events. In other words, real-time computation must include the option of selecting a response that will exist by the time at which the response is required. The computation time for the response can be reduced by storing a number of predefined responses at the expense of storage space. At the implementation, a reactive scheduling system may need to make trade-offs between computation time and storage space. In addition, further trade-offs between the solution quality and the amount of search required may needed.

Moreover, the frequency of reactive scheduling has some important effects. More reactive scheduling (with high frequency) will provide better service (user requests are quickly taken into account), better robustness to uncertainty, and more chances to update and improve the schedule than a single reactive scheduling. The drawback of this approach is that a shorter time interval for reactive scheduling may decrease the quality of the schedule. This is an interesting topic for further research.

Chapter 4

A Repair-Based Method for DCSPs

From Definition 15 we know that a Dynamic Constraint Satisfaction Problem (DCSP) is a sequence of static CSPs that are formed by constraint changes. In this sequence, the solution of one CSP may be invalidated by one or more constraint changes. Hence, a new solution must be generated for that CSP. To obtain a solution for such a CSP, it is possible to solve the CSP from scratch by a constructive method such as AC [1]. However, any solution obtained in this way might be quite different from a previous solution. In practical situations, large differences between successive solutions are often undesirable. For instance, the rescheduling of people working in a hospital in case two nurses are ill may result in too many changes to the old schedule; it means that many night shifts, weekend shifts and corresponding compensation days of all employees will be changed. Hence, for this scheduling problem, it is necessary to find a minimal change schedule with respect to the previously executing schedule.

4.1 Motivation

Several proposals to handle DCSPs have been presented in [?, ?, ?]. As pointed out in chapter 2, these proposals either are limited in a specific application domain or based on some particular assumptions which are not well-founded for changes caused by unexpected events. Our research is mainly motivated by the occurrence of unexpected events. The above-mentioned example of the hospital personnel is an illustrative guideline for the goals set.

Hence, to overcome the drawbacks of the previous methods, we proposed an

[1]In this chapter, AC stands for a backtracking search algorithm that applies arc-consistency on the yet unassigned variables when the algorithm starts or a variable has been assigned a value.

approach in our paper [?] that tries to repair the infringed solution of the current CSP instead of creating a completely new solution. The basic idea behind the approach is to move from a candidate solution, such as an infringed solution, that violates some constraints to a *nearby* solution that satisfy all constraints.

Whether a new solution found is nearby the infringed solution can be determined by a predefined *cost function* λ in the application domain. The cost may be the number of variables that are assigned a new value, the weighted distances between the new and the old values assigned to each variable, the importance of the variables that must change their value and so on. For conveniently dealing with DCSP, we formally define the nearby and *optimal* solution as follows.

Definition 20 *In a DCSP, the solution of the current CSP α_n is called the most nearby solution of the infringed solution α_i, if and only if the cost function $\lambda(\alpha_n, \alpha_i)$ is minimal. The most nearby solution is also called the optimal solution of the current CSP. Here the cost is defined as the number of variables that are assigned a new value.*

If we use other measures of the notion cost (e.g., see above), we explicitly mention this. We remark that it may happen that two or more solutions have the same minimal cost. A further discrimination is possible but for the moment we refrain from such a diversification.

As described in Section 3.4, local search is a solution method that moves from one candidate solution to a nearby candidate solution. However, it has a problem since there is no guarantee that it will find the most nearby solution. In fact local search may also wander off in the wrong direction. Another problem with using local search is the speed of the search process. Local search does not use something like constraint propagation to reduce the size of the search space. The search space is completely determined by the number of variables and the number of values that can be assigned to the variables.

It is our aim to develop a solution method that can systematically search the neighborhood of the infringed solution, taking advantages of the powerful constraint-propagation methods. In the following sections, we show that such a repair-based approach is possible.

4.2 Ideas and methodology

The problem we have to solve is the following. We have a CSP (V, D, C) and a complete assignment a that assigns values to all variables. The complete assignment a satisfied all constraints. But because of an incident some constraints have changed or new constraints have been added and a new CSP is formed. As a result the complete assignment a no longer satisfies all constraints in the new CSP, i.e., a becomes an infringed solution of the new CSP. Now it is our task to find a new

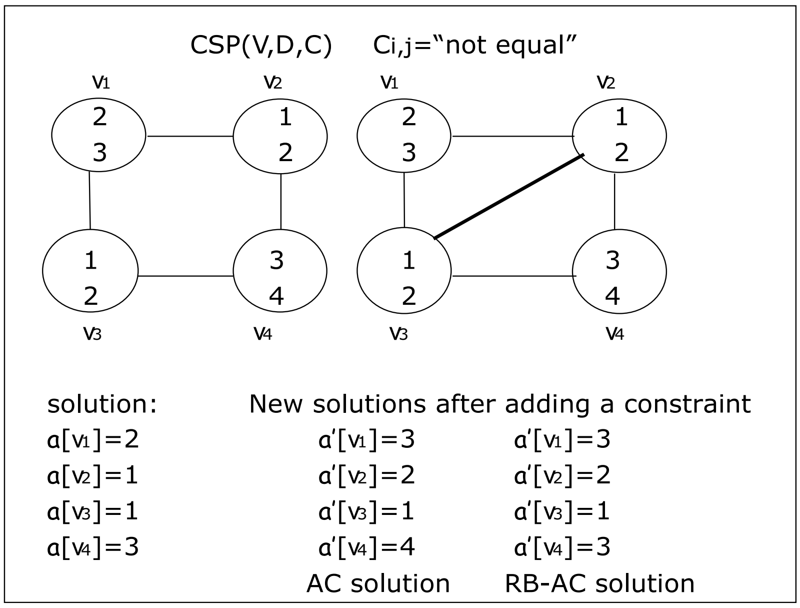


Figure 4.1: Basic idea of repair-based solving for a CSP

complete assignment a' that satisfies all constraints in the new CSP with a minimal cost.

4.2.1 Basic ideas

From Definition 20 it follows that if a' is a complete assignment that has a minimal number of different value assignments with a and satisfies all constraints in the new CSP, then a' is an optimal solution of the new CSP.

The example of Figure 4.1 illustrates the idea of repair-based solving a CSP.

Example The left graph of Figure 4.1 shows a constraint graph of a CSP instance, where v_1, v_2, v_3 and v_4 are four variables; $D_{v_1} = \{2, 3\}$, $D_{v_2} = \{1, 2\}$, $D_{v_3} = \{1, 2\}$ and $D_{v_4} = \{3, 4\}$ are the corresponding domains; each line linked between two variables denotes a *not equal* constraint. $a(v_1) := 2$, $a(v_2) := 1$, $a(v_3) := 1$ and $a(v_4) := 3$ is a complete assignment that satisfies all constraints. Now, assume that a new CSP is formed by adding a binary constraint (*not equal*) between v_2 and v_3. The corresponding constraint graph is showed in the right graph of Figure 4.1.

Consequently, the values assigned to the variable v_2 and v_3 are not consistent with this new added constraint. We use a set X to collect these variables, where $X := \{v_2, v_3\}$. So, at least one variable in the set X that violates the new added constraint must change its value assignment. In the new CSP, let us first perform the constraint propagation (enforcing arc-consistency). We found that no value assignment made by a is outside the new domain of each variable (obviously, no domain is changed after constraint propagation). Then, let us try to change one value assignment of a. If we select v_2 from X and assign it a new value $a(v_2) := 2$, the new added constraint is satisfied. However, the constraint between v_1 and v_2 is violated. After forward-checking arc-consistency, the current value assignment of v_1 is no longer in its domain. We use a set U to collect such variables ($v_1 \in U$). This means that we have to continue changing another variable's assignment, namely the value assignment of v_1. Since we are trying to change one value assignment of a, it is not allowed to change two value assignments of a in this iteration. Thus, we restore the initial value assignment of v_2 and the initial domain of v_1, select v_3 from the set X and assign it a new value $a(v_3) := 2$. The same situation arises as in changing the value assignment of v_2. As a result, all possibilities of changing one value assignment of a are exhaustively examined without success. Hence, we have to increase the number of new value assignments of a in the current CSP.

In order to find the most nearby solution such that all constraints are satisfied, a systematic exploration of the search space is required. In other words, if we fail to find a solution by changing the value assignment of one variable, we must look at changing the value assignment of two variables. If this also fails, we go to three variables, and so on. This process is the well-known *iterative deepening* strategy.

In the second iteration of solving the example of Figure 4.1, we focus on changing the value assignments of two variables. Let us select v_2 and assign it a new value, i.e., let be $a(v_2) := 2$ (repeat the work for changing the value assignment of one variable). Now v_1 is in the recalculated set U. Moreover, in this iteration we are allowed to change the value assignment of v_1 (i.e., let be $a(v_1) := 3$). It is easy to see that the new complete assignment a' made for v_1, v_2, v_3 and v_4 ($a'(v_1) := 3, a'(v_2) := 2, a'(v_3) := 1, a'(v_4) := 3$) satisfies all constraints in the new CSP. The algorithm returns success. According to the iterative deepening strategy, a' is also the most nearby solution of the new CSP (the initial assignments of v_3 and v_4 are not changed). Note that a constructive algorithm, which creates a solution from scratch, may find a solution with more assignment changes, namely 3 instead of 2 (see Figure 4.1).

4.2.2 The general case

Now, we return to the general case. Let X be the set of variables involved in the constraints that do not hold.

$$X = \bigcup_{c_{v_{i_1},\ldots,v_{i_k}} \in \mathcal{C}, c_{v_{i_1},\ldots,v_{i_k}}(a(v_{i_1}),\ldots,a(v_{i_k}))=\mathsf{false}} \{v_{i_1}, ..., v_{i_k}\}$$

Obviously, at least one of the variables in X must be assigned a new value. So, we can begin to investigate whether changing the value assignment of one variable in

X enables us to satisfy all constraints. If no such variable can be found then there are two possibilities.

- if
$$\bigcap_{c_{v_{i_1},\ldots,v_{i_k}} \in \mathcal{C}, c_{v_{i_1},\ldots,v_{i_k}}(a(v_{i_1}),\ldots,a(v_{i_k}))=\mathsf{false}} \{v_{i_1}, \ldots, v_{i_k}\} = \emptyset.$$
at least one of the other variables in X must also be assigned another value. Hence, we must investigate changing the value assignment of two variables in X.

- There is a value assignment to a variable v in X such that all constraints between the variables of X are satisfied, but for which a constraint $c_{v_{i_1},\ldots,v_{i_k}}$ with $v \in \{v_{i_1}, \ldots, v_{i_k}\}$ and $\{v_{i_1}, \ldots, v_{i_k}\} \not\subseteq X$, is not satisfied.

In both cases we must try to find a solution by changing the value assignment of two variables. In the former case it is sufficient to consider the variables in X for this purpose. In the latter case we must also consider the neighboring variables of X. The reason is that for any value assignment satisfying the constraints over X, there is a constraint over variables in X and variables in V-X that does not hold. We can determine the variables in V-X after assigning a variable in X a new value, by recalculating X and subsequently removing the variables that have been assigned a new value.

Briefly speaking, in order to find the most nearby solution, we should assign new values to variables in the set X. The number of variables that should be assigned a new value is increased gradually until we find a solution. Furthermore, the set of variables X that collect candidate variables for change is determined dynamically.

4.2.3 Improving performance efficiency

Considering many sets of variables to which we are going to assign a new value may have a high complexity. Successively, we give a quick calculation on the complexity of the method proposed. If we are going to change the values of n variables from a set X containing m candidate variables, we need to consider $\binom{m}{n}$ different subsets of X where each subset is a separate CSP. We call this number the first source of overhead. Since the number can be exponential in m (depending on the ratio between m and n) the number of CSPs we may have to solve is $O(2^m)$, where each CSP has a worst-time complexity that is exponential. This would make it impossible to use a repair-based approach.

There is another source of extra overhead in the search process. If we fail to find a solution changing the value assignments of n variables, we try for some larger value $n' > n$. Clearly, when changing the value assignments of n' variables we must repeat all the steps of the search process for changing the value assignments of n variables. This second source of overhead can, however, be neglected in comparison with the first source of overhead in the worst cases.

Constraint propagation can be used to speed up the search process in constructive solution methods by pruning the search space. We investigate whether it can also be used to avoid solving an exponential number of CSPs. If we could use constraint propagation to determine variables that must be assigned a new value, we may avoid solving a substantial amount of CSPs. In fact, we are able to do this by determining the domain values of the variable that are not allowed by the constraints, independent of the original assignment. If a variable was assigned a value that is no longer in its current domain, we know that it must be assigned a new value.

In the same way we can also determine a better lower bound for the number of variables that must change their current value assignments during the search process. Thereby we reduce the second source of overhead. The following example gives an illustration of how constraint propagation helps us to reduce both forms of overhead.

Example The left graph of Figure 4.2 shows a CSP instance, where v_1, v_2, v_3 and v_4 are four variables; $D_{v_1} = \{2,3\}$, $D_{v_2} = \{1,2\}$, $D_{v_3} = \{2,4\}$ and $D_{v_4} = \{3,2,4\}$ are the corresponding domains; each line linked between two variables denotes a *not equal* constraint. $a(v_1) := 2$, $a(v_2) := 1$, $a(v_3) := 4$ and $a(v_4) := 3$ is a complete assignment that satisfies all constraints.

Now, because of some incidents a unary constraint $c_{v_2} = (v_2 \neq 1)$ is added to the CSP (as shown on the right graph of Figure 4.2). If we enforce arc-consistency on the domains D_{v_1}, D_{v_2}, D_{v_3} and D_{v_4} by using the constraints in the original CSP and newly added constraint, we get the following reduced domains $D_{v_1} = \{3\}$, $D_{v_2} = \{2\}$, $D_{v_3} = \{4\}$ and $D_{v_4} = \{3\}$. From these reduced domains it immediately follows that two variables v_1 and v_2 must be assigned a new value in order to obtain a solution. So there is no point in considering which variables may need to change their value assignments nor investigating whether we can find a solution by changing one or two variables.

For a complicated CSP, the solution is not so straightforward after applying constraint propagation. However, based on the new value assignments made so far, we can try, by using the constraint propagation, to determine the other variables whose current value assignments lay outside of their current domain, and thus must also be assigned a new value.

4.2.4 The methodology

Based on the ideas explained, the examples given and many more examples investigated in our research, we developed a methodology for our investigation of repair-based approaches. The methodology relies on an appropriate representation (given in Chapter 2) and has as essential points: constraint propagation, systematic search and iterative deepening.

According to the methodology given above, we proposed an algorithm presented

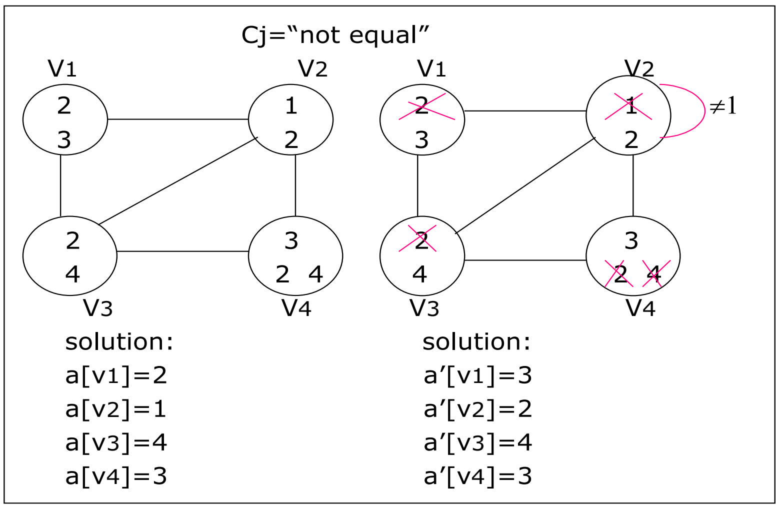


Figure 4.2: How constraint propagation reduces the overhead.

in the next section in which the ideas are implemented in combination with local search (repairing an infringed solution) and constraint propagation (enforcing arc-consistency). For convenience, this algorithm is denoted as RB-AC in this thesis.

4.3 Local search and constraint propagation

Let V be a set of variables, D an array of a domain for each variable, C a set of constraints and a an array containing a complete assignment. The RB-AC algorithm which consists of four procedures, $RB, solve, find$, and $assign$ is presented below, where V, D, C and a are global variables in all procedures with exception of RB. All arguments in the procedures are used both as input and output parameters. The RB-AC algorithm is used to solve the CSP (V, D, C) and to obtain a most nearby solution represented by the output a with respect to the infringed solution represented by the input a.

```
Procedure $RB(V, D, C, a)$
; returns success or failure
; $V$ a fixed set of variables
; $D$ the sets of allowed domain values of the variables in V; can be changed in the
; procedure, but will be unchanged after the return out of the procedure
; $C$ a fixed set of constraints on the variables V
; $a$ the assignment that is to be repaired; can be changed; will be a consistent
; assignment in case of success; will be the original assignment in case of failure
    save($a$);
    $X := conflict\_var(C, a)$;                (1)
    save($D$);
    constraint-propagation($V, D$);
    $U := \{v \in V \mid a[v] \notin D[v]\}$;      (2)
    $X := X \cup U$;                           (3)
    $n := max(1, |U|)$;                        (4)
    $u := |V|$;
    if ($X = \emptyset$)
       return success;
    end if;
    state = $solve(U, X, n, u)$;
    restore($D$);
    if (state = success)
       return success;
    else
       restore($a$);
       return failure;
    end if;
end Procedure RB.

Procedure $solve(U, X, n, u)$
; $U$ a subset of $V$ of variables that certainly need a new assignment
; $X$ a subset of $V$ of variables that probably need a new assignment, $X$ contains $U$
; $n$ the maximum number of new assignments allowed; *can change*
; $u$ is the greatest lower bound on the number of assignment changes that might
; lead to a solution given the current state of the search process (given the part of
; the search tree that has been investigated); *can change.*
    save($U, X$)
    while ($n \leq |V|$)
       if ($find(V, 1, U, X, \emptyset, n, u) = success$)
          return success;
       end if;
       $n := max(n + 1, u)$;                   (5)
       $u := |V|$;
       restore($U, X$);
    end while;
    return failure;
end Procedure $solve$.

Procedure $find(F, i, U, X, Y, n, u)$
```

```
; $F$ the subset of variables in $V$ that are allowed to get a new assignment (they did
; not yet get a new assignment)
; $i$ the count of the variable that needs a new assignment now ($i-1$ variables already
; got a new assignment)
; $U$ the set of variables that certainly need a new assignment
; $X$ the set of variables that probably need a new assignment ($X$ contains $U$)
; $Y$ the set of variables that is not allowed to have a new assignment
; $n$ and $u$ as in *solve*
     if ($U \neq \emptyset$)                          (6)
        $v$:= select-variable($U$);
        if ($assign(v, F - \{v\}, i, Y, n, u)$ = success)
           return success;
        end if;
     else
        while ($X \neq \emptyset$)                    (7)
           $v$ := select-variable($X$);
           $X := X - \{v\}$;
           if ($assign(v, F - \{v\}, i, Y, n, u) = success$)
              return success;
           end if;
           $Y := Y \cup \{v\}$;                       (8)
           $F := F \cup \{v\}$;                       (9)
        end while;
     end if;
     return failure;
end Procedure $find$.

Procedure $assign(v, F, i, Y, n, u)$
; $v$ a variable that needs a new assignment now
; $F$ the subset of variables in $V$ that are allowed to get a new assignment (they did
; not yet get a new assignment)
; $i$ the count of the variable that has to get a new assignment now and will get this
; new assignment if possible
; $Y$ a set of variables that is not allowed to get a new assignment
; $n$ and $u$ as in procedure solve
     save($D[v], a[v]$);
     state := failure;
     $D[v] := D[v] - \{a[v]\}$;
     while ($D[v] \neq \emptyset \wedge state = failure$)
        $d$ := select-value($D[v]$);
        $D[v]$ :=$D[v] - \{d\}$;
        $a[v] := d$;                                  (10)
```

```
    save($D[v^*],\ \forall v^* \in F$);
    constraint-propagation($F, D$);
    if ($\forall v^* \in F, D[v^*] \neq \emptyset$)
       $U := \{v^* \in F \mid a[v^*] \notin D[v^*]\}$;        (11)
       $X := U \cup conflict\_var(C, a)$;          (12)
       $j := max(1, |U|)$;
       if ($X = \emptyset$)
          return success;
       else
          if ($i + j \leq n$)
             if ($U \cap Y = \emptyset$)
                state := $find(F, i+1, U, (F \cap X) - Y, Y, n, u)$
                if (state = success)
                   return success;
                end if;
             end if;
          else
             $u := min(u, i + j)$;                   (13)
          end if;
       end if;
    end if;
    restore($D[v^*], \forall v^* \in F$);
  end while;
  restore($D[v], a[v]$);
  return failure;
end Procedure $assign$.
```

Where: $conflict\_var(C, a) := \bigcup_{c_{v_{i_1},\cdots,v_{i_k}} \in C, c_{v_{i_1},\cdots,v_{i_k}}(a(v_{i_1}),\cdots,a(v_{i_k}))=false} \{v_{i_1}, \cdots, v_{i_k}\};$

The algorithm RB-AC is activated by calling the procedure $RB(V, D, C, a)$ which starts by identifying the current problem. Assume a is an infringed solution of the given CSP (V, D, C), so a violates one or some of the constraints in C. Consequently, at least some of the variables involved in these violated constraints need to change their value assignments. The function $conflict\_var(C, a)$ determines the variables involved in such constraint violations, and the set X (RB-AC algorithm, line (1)) collects these variables. Moreover, the constraint propagation[2] is applied on the original domains of all variables to determine the variables that must be assigned a new value, i.e., the variables of which the current value assignments of a lay outside their domains after constraint propagations. These variables are collected in the set

[2]arc-consistency

U (RB-AC algorithm, line (2)). As a result, the new set X, i.e., the set of all variables which may be assigned a new value (RB-AC algorithm, line (3)), is updated. Note that often only a subset of X need to be assigned a new value. Thereafter, the initial number of variables n that are to be assigned a new value, is set at $max(1, |U|)$ (RB-AC algorithm, line (4)). The constant $|V|$ delimits the maximum number of changes on the value assignments of the infringed solution a. The variables n and u will be updated when the algorithm proceeds. If X is empty, RB immediately returns $success$. Otherwise, whether RB returns $success$ or $failure$ is dependent on the outcome of the procedure call $solve(U, X, n, u)$.

The procedure $solve(U, X, n, u)$ carries out an iterative deepening search strategy. While $n \leq |V|$, $solve(U, X, n, u)$ calls the procedure $find(V, 1, U, X, \emptyset, n, u)$. If $find(V, 1, U, X, \emptyset, n, u)$ does not return success, $solve$ updates n (RB-AC algorithm, line (5)) to the new maximal number of variables that may be assigned a new value and then restarts $find(V, 1, U, X, \emptyset, n, u)$. This processes will be iteratively performed until $find$ returns $success$ or $n > |V|$ without $success$.

The procedure $find(F, i,\ U, X, Y, n, u)$ selects a variable to be assigned a new value. If the set U is not empty (RB-AC algorithm, line (6)), $find$ selects a variable v from the set U since the variables in U must be assigned a new value. Subsequently, the procedure $assign(v, F, i, Y, n, u)$ is called. Otherwise, it selects a variable v from the set X then calls the procedure $assign(v, F, i, Y, n, u)$. If the call of $assign(v, F, i, Y, n, u)$ fails and the variable was selected from the set U, $find$ returns $failure$. If the variable was selected from the set X and $i < n$, $find$ will try other variables in the set X one by one until a call of $assign(v, F, i, Y, n, u)$ returns $success$ or the set X becomes empty (RB-AC algorithm, line (7)). The variable set Y is used to represent the CSP variables for which new assignments have been tried without $success$ in the same $find$ call.

The procedure $assign(v,\ F, i, Y, n, u)$ assigns a new value to the selected variable v and carries out constraint propagation to evaluate the effects of the assignments made so far. The sub-procedure constraint-propagation(F, D) applies forward checking and full look-ahead arc-consistency on the future variable set F given the new assignments made to the past variables and the current variable v. For the case more variables need to be assigned a new value is determined by recalculating the sets U (RB-AC algorithm, line (11)) and X (RB-AC algorithm, line (12)). If $X = \emptyset$, the problem is solved and $success$ is returned. Otherwise, $assign$ either recursively calls the procedure $find$ ($when\ i < n$) or updates u (RB-AC algorithm, line (13)) and backtracks to assign another value for the selected variable v. If the set X is still not empty after the domain of the selected variable becomes empty, $assign$ returns $failure$.

In summary, RB-AC finds an optimal solution by systematically searching the neighborhood of the infringed solution with an iterative deepening approach. If RB-AC fails to find a solution through an exhaustive search with changing the value assignments of n variables, it tries to change the value assignments of n' variables with $n' > n$ (n' is determined by line (5) and line (13) in RB-AC algorithm). If this also fails, it goes to n'' variables with $n'' > n'$, and so on until the value of $|V|$ is reached. If RB-AC then fails there is no complete assignment a' possible (no solution exists).

4.4 Supportive theorems

The following theorems state that, for a given CSP (V, D, C) and a complete assignment a, the RB-AC algorithm either finds the solution most nearby a if a solution exists, or proves that no solution exists.

Theorem 2 *Let V be a set of variables with domains D. Let C be a set of constraints on V. Let a be a complete assignment of the variables in V. If a consistent assignment A exists for the CSP(V, D, C) then the procedure call RB(V, D, C, a) will return success and the assignment a is changed into a consistent assignment such that for a minimal number of variables the original assignments are changed. If a consistent assignment does not exist then the procedure call RB(V, D, C, a) will return failure and the original assignment is unchanged.*

Proposition 1 *Let V be a set of variables with domains D. Let C be a set of constraints on V. Let U be a subset of V of variables that certainly need a new assignment. Let X be a subset of V of variables that probably need a new assignment (X contains U). Let n be an integer such that initially $n \geq |U|$ and $n \leq u$. Then the call to the procedure $solve(U, X, n, u)$ will result in success if a new assignment exists with n changes of variables with respect to the original assignment, otherwise this call will result in failure. In case of success the assignment a is changed into a consistent assignment. The variables in the set U have got a new assignment, and possibly some more variables have got a new assignment. During the procedure, n is altered into the minimal number of assignment changes needed for a consistent assignment or into $|V| + 1$ in case of failure.*

Proposition 2 *Let F be the subset of variables in V that are allowed to get a new assignment (they did not yet get a new assignment). Let i be an integer, $1 \leq i \leq n$, such that $i - 1$ is the number of variables that already got a new assignment. Let U be a subset of variables that certainly need a new assignment. Let X be the set of variables that probably need a new assignment (X contains U). Let Y be the set of variables that is not allowed to have new assignments. Let n be an integer denoting the maximally allowed total number of variables that will have new assignments and let u be the greatest lower bound on the number of assignment changes that might lead to a solution given the current state of the search process, where $n \leq u$. Then the procedure call $find(F, i, U, X, Y, n, u)$ will result in success if the assignment a adapted by the assignment on the $i-1$ already newly assigned variables can be adapted to a consistent assignment with at most $n - i + 1$ new assignments to variables in F*

but not in Y. Otherwise the call will result in failure. If the procedure call results in success then the assignment a is changed into a consistent assignment such that all variables in U got a new assignment and the total number of changed variables with respect to the original assignments is at most n.

Proposition 3 *Let v be a variable that needs a new assignment now. Let F be the subset of variables in V that are allowed to get a new assignment (they did not yet get a new assignment). Let i be an integer, $1 \leq i \leq n$, such that $i - 1$ is the number of variables that already got a new assignment. Let Y be the set of variables that are not allowed to have new assignments. Let n be an integer denoting the maximally allowed total number of variables that will have new assignments and let u be the greatest lower bound on the number of assignment changes that might lead to a solution given the current state of the search process, where $n \leq u$. Then the procedure call $assign(v, F, i, Y, n, u)$ will result in success if the assignment a adapted by the assignment on the $i - 1$ already newly assigned variables can be adapted to a consistent assignment with a new assignment to the variable v and at most $n - i$ new assignments to variables in F but not in Y. Otherwise the call will result in failure. If the procedure call results in success then the assignment a is changed into a consistent assignment such that all variables in U got a new assignment and the total number of changed variables with respect to the original assignments is at most n. If the call results in failure u will denote the minimum required number of new assignments and the assignments of the variables in $F \cup \{v\}$ are the original values.*

Theorem 2 expresses the termination, correctness, completeness and optimality of RB-AC algorithm. We will prove that Theorem 2 is a consequence of Proposition 1, and Proposition 1 is the consequence of Proposition 2.

First, we give the proof for propositions 2 and 3. Then, we give the proof for proposition 1 and Theorem 2.

Proof of Proposition 2 and 3.

Let us consider proposition 3. Suppose all preconditions for the call $find(F, i, U, X, Y, n, u)$ are fulfilled. We show that the preconditions for the calls $assign(v, F - \{v\}, i, Y, n, u)$ during the procedure $find$ are fulfilled too.

Based on $i - 1$ newly assigned variables, $find$ calls $assign$ to change the assignment of the ith variable. (1) When the parameter U of $find$ is not empty, the ith variable v must be selected from U. Namely, the variables in U must get new values. For the call of $assign$, the parameters i, Y, n, u and the set F are the same as those of $find$. So, the preconditions of Proposition 3 are fulfilled when a $find$ calls $assign$ in this case. (2) When the parameter U of $find$ is empty, the ith variable v is selected from X. In this case, if a call of $assign$ returns success, $find$ returns success too. After trying every variable in X without success by calling $assign$, $find$ will returns failure. So, $find$ calls $assign$ at most $|X|$ times. For all calls of $assign$, the parameters i, n and the set F are the same as those of $find$. However, the parameters Y, u are changed dependent on the previous calls of *assign*. Any previously tried variable v by a call of *assign* without success will be added to the set Y, since for any new assignment of v, there does not exist a solution requiring no more than n assignment changes. Such calls of *assign* will update the value of u. So, the preconditions of Proposition 3 are fulfilled too, when each *find* calls *assign*.

Hence, if proposition 3 is proven, we can conclude from the described results for the calls of *assign*, that in the corresponding cases proposition 2 also holds.

Let us consider proposition 3.

Suppose all preconditions for the call $assign(v, F, i, Y, n, u)$ are fulfilled. Procedure *assign* adapts the assignment of the ith variable v of a based on $i-1$ already newly assigned variables. We first prove that the proposition holds when *assign* returns success, and afterwards we prove that the proposition holds for *assign* returning failure.

assign returns success if there is a value in the current domain of v which does not result in an empty domain for one of the future variables in F after constraint propagation and if one of the following two cases occurs: (1) the new set X is empty, or (2) a recursive call to *find* is made and returns success. Note that since constraint-propagation imposes arc-consistency on the future variables F, any assignment to v is consistent with the $i-1$ already newly assigned variables.

- In case (1), the sets U and X are empty. This means that for each variable in F, its assignment of a is still in its current domain and is not involved in any constraint violation. a is thus adapted as a consistent assignment with i newly assigned variables. Since $i \leq n$, Proposition 3 holds.

- In case (2), the new set X is not empty. The set U (containing the variables of which the current assignments are not in their current domain) may not be empty too. Moreover, $i + j$ is less than or equal to n and $U \cap Y$ is empty. The first condition states that the number of necessary assignment changes in F will be less than or equal to $n - i$; the call to *find* adapts at least $j = |U|$ assignments in F. The second condition states that no variable that must be assigned a new value belong to the set of variables that may not be assigned a new value because they have already been tried. Therefore, a recursive call to *find* is made and this call returns success. Clearly, when calling $find(F, i+1, U, (F \cap X) - Y, Y, n, u)$, (i) F is the set of future variables that have not been assigned a new value, (ii) i is the number of variables that have been assigned a new value since v has been assigned a new value, (iii) X contains the variables involved in a constraint violation and $(F \cap X) - Y$ contains those future variables in X that may receive a new value because they have not been tried before, (iv) Y contains the variables that may not receive a new value because they have been tried before, (v) n is the maximum number of assignment changes and (vi) u is the greatest lower bound on the number of assignment changes that might lead to a solution given the current state of the search process so far. Since $find$ returns success, at most n variables have received a new value, including the variables in U, a contains the modified assignments and all constraints hold given a. Hence, Proposition 3 holds in case it returns success.

assign returns failure after trying every value in v's current domain $D[v]$ (through while loop) without success. There are four reasons why a new assignment may not lead to success: (1) the domain of a future variable (in F) becomes empty after constraint propagation; (2) no future variable's (in F) domain becomes empty after constraint propagation, but the number $|U|$ of variables that must be assigned a new value exceeds the number of allowed assignment changes $n - i$; (3) a variable that must receive a new value has already been tried before without success, $U \cap Y \neq \emptyset$; and (4) the call to $find$ returns failure. In case 2, the value of u (the greatest lower bound on the number of assignment changes that might lead to a solution given the current state of the search process) is updated, at least $i + j$ assignment changes are required given the changes made so far. In case 4, no solution exists requiring no more than n assignment changes. The parameter u of $find$ will contain an upper bound on the minimum number of required assignment changes needed if the call to $find$ had to succeed. Hence, Proposition 3 also holds in case it returns failure.

So, Proposition 3 holds whenever proposition 2 holds, and Proposition 2 holds whenever proposition 3 holds.

In order to complete the proof we remark that for both procedures holds that they only can succeed or fail when $1 \leq i \leq n$. Furthermore, if $find$ is called with a value i equal to i_0, then it will call *assign* with the same value of i equal to i_0. Then either *assign* needs a call to $find$ with a value of i equal to $i_0 + 1$ or *assign* does not need such a call. In the latter case *assign* results in a success or failure. In the former case we can use an induction argument: These recursive calls will finish either with success or failure for some i, say i_1, with $i_1 \leq n$. For that last value of i *assign* terminates, and so does the calling $find$, and so on. So in fact the induction proves that since the calls terminate when $i = i_1$, they will terminate for $i = i_1 - 1, i_1 - 2, \cdots, 1$. □

Now, we give the proof for proposition 1.

Proof of Proposition 1.

Suppose all preconditions for the call $solve(U, X, n, u)$ are fulfilled. Then for any value of $n \leq |V|$, the parameters of $find(V, 1, U, X, 0, n, u)$ satisfy the conditions of Proposition 2. Therefore, if the call of $find$ succeeds, then a contains a solution requiring at most n assignment changes with respect to the original infringed solution. If a call of $find(V, 1, U, X, 0, n, u)$ returns failure, no such solution exists for the current parameter values of n and u, and *solve* tries to find a solution requiring more assignment changes by calling $find(V, 1, U, X, 0, n, u)$ with new parameter values of n and u. Hence *solve* contains at most $|V|$ iterative calls of $find$ and returns success if and only if there exists a solution for the current CSP.

For each call of $find(V, 1, U, X, 0, n, u)$, the sets V, U, X are the same as those in *solve*. If one call of $find$ returns success, the variables in U get new values. To complete the proof of the proposition, we have to show that each time a call to $find(V, 1, U, X, 0, n, u)$ is made, *solve* "knows" that no solution requiring less than n assignment changes exists. We can distinguish three situations. (1) In the first call to $find$, $n = |U|$ (the variables that must be assigned a new value). (2) If n has been increased with 1 after the previous call, then no solution requiring no more than $n - 1$ assignment changes exists. (3) If n has been increased with $k > 1$ after the previous calls, then this is because of the value of u after the previous calls. According to Proposition 2 (and Proposition 3), no solution exists requiring less than u assignment changes. Hence, if *solve* succeeds, no solution requiring less than n assignment changes exists. □

The proof of Theorem 2 is as follows.

Proof of Theorem 2.

Procedure RB initializes the parameters used by *solve*. The values of these parameters satisfy the conditions of Proposition 1. If $X = \emptyset$, the original assignment a satisfies the (changed) set of constraints. Therefore, RB returns success and Theorem holds in this case.

If $X \neq \emptyset$, X contains the variables involved in a constraint violation and U contains the variables that must be assigned a new value. Hence the conditions of Proposition 1 are satisfied. According to propositions 1 *solve* returns success if and only if a solution for the CSP exists. Moreover, in case of success, the call to *solve* changes the assignment of a minimum number of variables in V. □

In the Example of Figure 4.2, we have illustrated that RB-AC algorithm reduces the search overhead by applying constraint propagation. Now, we present a result in which all overhead can be completely eliminated in case unary constraints are added to an instance of a CSP. Since some unexpected events in practical situations, such as the unavailability of machines or the lateness of supplies, can be described by unary constraints, this result is practically significant.

Proposition 4 *Let (V, D, C) be a CSP containing only unary and binary constraints and let the assignment a be a solution.*

If a new CSP is formed by adding only unary constraints to the original constraint set, then using node-consistency *in the procedure* RB *and* forward checking *in the procedure 'assign' avoid considering more than one subset of X.*

Proof Enforcing node-consistency guarantees that we know all variables of which the assignments in a are inconsistent with the new added unary constraints. So, $X = U$ in *RB* procedure.

Having assigned some variable v a new value, some of the binary constraints $c_{v,w}$ may no longer hold. Since we apply forward checking, values in the domain of each variable w that are inconsistent with the new value assigned to v will be removed from the domain of w. If the constraint $c_{v,w}$ does not hold given the current value of w, this value will not be in the domain of w after forward checking. So again, $X = U$.

Hence, we always consider only one subset of X in which every variable assignment should be changed. $\square$

The above proposition implies that repairing a solution after adding unary constraints to a CSP does not have a high complexity. Using proper constraint-propagation techniques can bring the complexity close to $O(T)$, where T is the complexity of solving the CSP from scratch. If, however, binary constraints are added, the situation changes. If the current solution does not satisfy the added binary constraint, one of the two variables of the constraint must be assigned a new value, and possibly both. Since we do not know which one, both possibilities will be investigated until a new solution is found. This implies that the repair time will double with respect to adding a unary constraint.

4.5 Approximate algorithms

From Theorem 2, RB-AC theoretically can find the most *nearby* (optimal) solution for a CSP (if such a solution exists) in a DCSP. Moreover, if the new CSP is the result of adding a unary constraint, the time complexity of RB-AC is not much higher than solving the new CSP from scratch. However, when a new CSP is formed by adding or changing several n-ary ($n \geq 2$) constraints, and this happens quite often in a practical situation, using RB-AC to find a minimal change solution is much harder than using a constructive algorithm to generate an arbitrary solution from scratch. The time complexity of RB-AC then increases drastically. The constraint-propagation techniques integrated in RB-AC do not reduce the time complexity of RB-AC. To solve the CSPs, RB-AC may not find an optimal solution over a reasonable period of time. The reason is that the search efficiency of RB-AC is affected when we intend to obtain an optimal-solution stability, i.e, maintaining as many as possible variable assignments. Hence, we face an important trade-off between search efficiency and solution optimality.

By examining the RB-AC algorithm more closely, we found that a significant portion of the search effort in RB-AC was consumed by additional overhead that the constructive algorithm AC does not have. In our paper [**?**], we provided an thorough analysis of the time complexity of the RB-AC algorithm. The analysis will be introduce in Subsection 4.5.1. Then, in the Subsections 4.5.2 and 4.5.3 we deal with two approaches that balance the search efficiency and solution optimality in such a way that a part of the additional overhead is avoided and a near-optimal solution is obtained within a reasonable period of time. The first idea is to do a binary search on a maximum number of variables (called the BS algorithm), the second idea is to restart the search processes a number of times and take the best one (called the RS algorithm).

4.5.1 Time complexity of RB-AC algorithm

Unlike a search algorithm that constructs a solution from scratch, RB-AC uses an iterative deepening strategy and searches through the subsets of variables, i.e., the subsets of the set X. Since the number of such subsets is exponential, search through the subsets can have a significant influence on the time complexity of RB-AC. Below, we analyze the resulting time complexity of RB-AC.

Let us assume that RB-AC changes the value assignment of N ($N \leq |V|$) variables and obtains the optimal solution. For the sake of dealing with the worst cases in the execution of RB-AC, we also assume that the set U (RB-AC algorithm, lines (2) and (11)) is always empty. Moreover, the set X (RB-AC algorithm, lines (3) and (12)) at depth i contains at most $|V| - i + 1$ variables and the maximal domain size of any variable in X is $d = max_{v \in V} |D_v|$.

The maximum number of nodes (M) that RB-AC may examine is:

$$M \leq |V| \cdot d + (|V| - 1)^2 \cdot d^2 + \cdots + (|V| - N + 1)^N \cdot d^N \tag{4.1}$$

We have $(|V| - i + 1)^i < (|V| - i)^{i+1}$ for every $i = 0, 1, \cdots, N$ ($N \leq |V|$). Hence,

$$M \leq (|V| - N + 1)^N \Sigma_{i=1}^{N} d^N = N(|V| - N + 1)^N d^N \tag{4.2}$$

Let be $K = |V| - N + 1$. From Formula (4.2), it can be inferred that the time complexity of RB-AC for changing the value assignment of N variables is $O(K^N \cdot d^N)$ in the worst case. In comparison with $O(d^N)$, which is the worst-case complexity of the constructive method AC that solves the CSP from scratch, the factor K^N is the main cause of the additional overhead introduced by the RB-AC search processes.

The constraint propagation reduces the contribution of the factors K^N and d^N in RB-AC as well as the factor d^N in the constructive algorithm AC. However, when a CSP is modified by adding or changing several n-ary ($n \geq 2$) constraints, the constraint propagation does not provide sufficient help for improving the performance of RB-AC. Moreover, the heuristics on variable or value selection do not exert any influence on the exhaustive search at each $n(n < \textit{minimal cost})$ depth in RB-AC. Hence, we have to investigate new approaches that are able to reduce the time complexity of RB-AC. In the following subsections, we introduce two approximate algorithms which were first proposed in our paper [**?**] that modify the RB-AC algorithm by balancing its solution optimality and its time complexity. They are the BS and RS algorithms.

```
BS (V,D,C,a)
  $lb := 1, ub := |V|$;
  while($ub \neq lb$) do
    $n1 := (ub + lb)/2$;
    if($solve(V, D, C, a) = success$) then
      cache solution ;
      $nd := new\_cost^*$;
      if($nd < ub$) then $ub := nd$;
      else $lb := (ub + lb)/2$;
    else $lb := (ub + lb)/2$;

Procedure $solve(V, D, C, a)$
  $X := conflict\_var$(C,a);
  constraint-propagation ($V$);
  $U := \{v \in V | \ a[v] \notin D[v]\}$;
  $X := X \cup U$;
  $n := n1$;
  if ($n \leq |V| \wedge BTN^* < upbound$) then
    if($find(V, 1, U, X, \emptyset) = success$)
    then return $success$;
  return $failure$;

Procedure find($F, i, U, X, Y$)
  $\{\cdots\cdots\cdots\cdots\cdots\cdots\}$;

Procedure assign($v, F, i, Y$)
  $\{\cdots\cdots\cdots\cdots\cdots\cdots\}$;
```

*where the *new_cost* is the cost of new solution; BTN is the number of backtrack steps.

Figure 4.3: BS algorithm.

4.5.2 The BS algorithm

The first approach, denoted as BS, uses binary search on the maximum cost (the maximum number of variables that may be assigned a new value). In this approach, the iterative deepening processes of RB-AC are replaced by a binary search on the maximum depth that the search processes might proceed. Moreover, BS sets an upper bound for the number of backtracks to prevent consuming too much time on one estimated maximum depth. Thus, for each estimated maximum depth, either a new solution is found or the upper bound of backtracks is reached without a new solution. Furthermore, since BS gives up the exhaustive search on each estimated search depth, the heuristics on variable and value selection become applicable. We use 'most-conflicts' as a variable-ordering heuristic and the 'mini-conflicts'[**?**] as a value-ordering heuristic in the implementation.

The pseudo codes of BS algorithm that are different from that of the RB-AC algorithm are presented in Figure 4.3. The procedures *find* and *assign* in BS are almost the same as they are in RB-AC except for adding backtrack upper-bound limitations in both procedures, without using n and u as parameters, and leaving out line (13) in RB-AC algorithm. BS works as follows. First, the number of the variables in the CSP is taken as the upper bound ub and 1 as the lower bound lb of the maximum depth. Second, BS tries to solve the problem by proceeding at most $(lb + ub)/2$ depths in the modified RB-AC. If it produces a solution and the *cost* of the solution is less than ub, this *cost* is taken as new ub. Otherwise, lb becomes $(lb + ub)/2$. When lb and ub are equal, the work is done. Third, the remaining solution is taken as the approximate optimal solution of the new CSP. If the final *ub* is equal to the number of variables in the CSP, the solution produced by BS may not be better than the one produced by a constructive algorithm AC.

If RB-AC finds a solution at depth N, it may search every depth between 1 and N. BS only searches through $O(logN)$ depths. From Formula (4.2) it follows that the time complexity of BS is still $O(K^N \cdot d^N)$ in the worst case. The actual reduction on the time consumption in BS is reached by setting the upper bound of backtracks for each estimated maximum depth. We may conclude that the BS approach avoids some repetition of work in the iterative deepening strategies of RB-AC and makes variable-selection and value-selection heuristics useful. But, it does not really reduce the time complexity of RB-AC.

4.5.3 The RS algorithm

The second approach avoids to traverse every variable in the set X (RB-AC algorithm, lines (3) and (12)). It is denoted as RS (Restart Search processes) and is explored below. RS randomly selects one variable from the set X in the procedure $find$ in case of an empty set U. Each time, a new set X (RB-AC algorithm, line (12)) is determined after the selected variable is assigned a value from its domain. If RS does not find a solution by changing the assignment of the selected variable, no other variable from the same set X is tried. Instead, RS backtracks to the variable selected in the previous search step and tries a new value for that variable. Moreover, the iterative deepening process is replaced by a depth first search process by setting the maximum depth equal to the number of variables in the problem ($n = |V|$ in RB-AC algorithm, line (4)) or the *cost* (depth) of the best solution found so far. The latter is called 'search-depth adaptation'. These two changes make the search processes of RB-AC similar to the search processes proposed by Verfaillie and Schiex [**?**]. The worst-case time complexity of our search processes is reduced to $O(d^N)$.

The pseudo codes of RS algorithm are presented in Figure 4.4. Due to the similarity with RB-AC, the pseudo codes of the procedures $find$ and $assign$ in RS are omitted in Figure 4.4. Only a few modifications on these two procedures are made. We mention them here. With reference to the RB-AC algorithm, in the procedure $find$ of RS, *while* (RB-AC algorithm, line (7)) is changed to if and the lines (8) and (9) (RB-AC algorithm) are left out; in the procedure $assign$ of RS, line (13) is left out.

In order to obtain a near-optimal solution, RS restarts the search process a number (RN) of times and caches the solution that has the minimal *cost* based on the solutions found so far. In each restart, backtracks are allowed and the upper bound of backtracks is set to prevent consuming too much time on one restart. A restart happens when a search process either finds a solution or reaches the upper bound of backtracks without a solution. Because the restart of the search is combined with a randomized variable-selection strategy, the probability of following the same search path more than once is rather small since the number of the possible paths is usually quite large. Hence, a different solution will be found in each restart. Furthermore,

```
RS (V,D,C,a)
  $cost := |V|$;
  $n1 := cost$;
  for($i := 1; i < RN; i := i + 1$) do
    if($solve(V, D, C, a) = success$) then
      if($new\_cost^* < cost$) then
        cache solution;
        $cost := new\_cost$;
        $n1 := cost - 1$;        (1)

Procedure $solve(V, D, C, a)$
  $X := conflict\_var$(C,a);
  constraint-propagation ($V$);
  $U := \{v \in V |\ a[v] \notin D[v]\}$;
  $X := X \cup U$;
  $n := n1$;
  if ($n \leq |V| \wedge BTN^* < upbound$) then
    if($find(V, 1, U, X, \emptyset) = success$)
    then return $success$;
  return $failure$

Procedure find($F, i, U, X, Y$)
  $\{\cdots\cdots\cdots\cdots\cdots\cdots\}$;

Procedure assign($v, F, i, Y$)
  $\{\cdots\cdots\cdots\cdots\cdots\cdots\}$;
```

*where the *new_cost* is the cost of new solution; BTN is the number of backtrack steps.

Figure 4.4: RS algorithm.

selecting a variable from the set X makes RS change the value assignment of the variables that are in the neighborhood of the infringed solution.

In this approximate approach, the execution efficiency and solution optimality are closely related to the following parameters: the number of restarts, the value-selection ordering, the upper bound on the number of backtracks, and the search-depth adaptation.

Obviously, more restarts will produce better solutions as well as an increase in time consumption. When the upper bound of backtracks is quite large, we do not adapt the search-depth (line (1) is left out in Figure 4.4) to the minimal *cost* based on the solutions found so far. If we did so, we would cause more backtracks in easy CSPs making them much harder to solve. As a result the time required to achieve a near-optimal solution increases considerably. However, when the upper bound of backtracks is relatively small, the search-depth adaptation is applicable. The reason is that if the adaptation brings about more backtracks than the upper bound of backtracks allowed, the search will immediately restart. Thus, the time consumption of each restart will be controlled to an acceptable amount and the RS will work in a way similar to the *branch-and-bound* search algorithm. An extreme circumstance of this approach is that the search process immediately restarts when a dead-end is encountered. This implies that a non-backtracking (i.e., backtrack-free) approach is applied.

As pointed out in Subsection 4.5.1, the value-selection heuristics do not improve the efficiency of the original RB-AC. In RS, however, it may help to find a solution more quickly in each restart. For this reason, the mini-conflict value selection [**?**]

and random value selection are examined in the implementation.

By doing a large number of experiments, we have tried to find out the proper parameter combination in RS to balance the execution efficiency and the solution optimality. We conclude that it is crucial to use a sufficiently large number of restarts and a moderate number of backtracks for a general CSP. If the CSP has looser constraints, the mini-conflict value selection should be applied in RS. If the CSP has tighter constraints, the random value selection or the mini-conflict value selection can be applied arbitrarily. Since it is in practice usually difficult to determine whether a given CSP has looser or tighter constraints, the mini-conflict value selection is preferred.

If the mini-conflict value-selection heuristic is used, search-depth adaptation and no search-depth adaptation can be applied arbitrarily for the CSPs with looser constraints. However, if CSPs have tighter constraints, the search-depth adaptation should be applied. Again, since it is in practice usually difficult to determine whether a given CSP is looser or tighter constrained, the search-depth adaptation is preferred.

In summary, whether a general CSP has looser or tighter constraints, using a sufficiently large number of restarts, a moderate number of backtracks, mini-conflict value selection and search-depth adaptation in RS is regarded as the best parameter combination for getting a near optimal solution.

4.5.4 Empirical studies

In this subsection, we provide an overview of our experiments concerning the approaches discussed in the previous subsections. To guarantee the generality, we assume that a randomly generated CSP is located in the sequence of a DCSP. The infringed solution of a CSP is a randomly-generated complete assignment of its variables.

We use a four-tuple $(\overline{n}, d, p_1, p_2)$ to generate CSP instances, where $\overline{n}$ denotes the number of variables and d denotes the domain size of each variable; p_1 and p_2 are two probabilities. They are used to generate randomly the binary constraints among the variables. Here, p_1 represents the probability that a constraint exists between two variables, and p_2 represents the conditional probability that a pair of values in the domains of the two variables is consistent. Therefore, the bigger the p_1 value, the more binary constraints exist among the variables of the generated CSPs (the CSP is tighter constrained). For a given value of p_1, the bigger the p_2 value, the more values are consistent in the domains of the two variables (the constraint between two variables is looser). When p_1 and p_2 take values from 0 to 1 with step 0.1, the generated CSP instances include three distinct classes: (1) the class of the least constrained ones (i.e., a solution is quickly found), (2) the class of the most constrained ones (an inconsistency is quickly detected) and (3) the intermediately constrained ones (it is difficult to establish a consistency or an inconsistency).

In the experiments, the initial complete assignment for the generated CSP instance is made by assigning a random value to each variable. The approaches discussed in the previous sections are tried to find a solution nearby such a complete assignment that is viewed as an 'infringed' solution of the considered CSP. This presupposition represents the worst possible cases for the repair-based algorithm, since many constraints will be violated, many variables will be involved in the violated constraints, and there will be no fixed pattern for the constraint violations.

Corresponding to the four-tuple $(\overline{n}, d, p_1, p_2)$, CSP instances were generated by setting $\overline{n}$:=20, d:=10; and $\overline{n}$:=40, d:=20. The choice for $\overline{n}$:=20 and d:=10 was motivated by the fact that RB-AC can still find the optimal solution within a reasonable period of time with these two parameter values. Choosing $\overline{n}$:=40 and d:=20 is to compare various approximate approaches in solving large-scale CSPs, since the original RB-AC is usually not able to solve the instances generated with these two parameter values within a reasonable period of time.

We show some of the experimental results for CSP $(20, 10, p_1, p_2)$ by the Figures 4.6 to 4.9, in which p_1 takes the values 0.3 (Figures 4.6 and 4.8) and 0.7 (Figures 4.7 and 4.9). These values of p_1 are examples of CSP with a low and high number of constraints respectively; p_2 takes the values from 0.1 to 1.0 with step 0.1. These values of p_2 cover the whole range of constraints from loose to tight ones. For each pair of values of p_1 and p_2, 10 instances are created to be solved by the algorithms. The figures present the solution quality and the time consumption of four algorithms, namely the constructive algorithm AC (denoted as FC in the Figures), the original repair algorithm RB-AC (denoted as RB in the Figures), and the approximate algorithms RS and BS. The Figures 4.6 and 4.7 exhibit the average distance to the randomly-generated 'infringed' solution. The Figures 4.8 and 4.9 exhibit the average time needed to find a solution to one problem instance. All experiments in this section are performed on a Pentium-450MHz PC with 128.0MB RAM and all algorithms use forward-checking and full-lookahead arc-consistency for constraint propagation. The upper bound of backtracks in RB-AC is 10,000,000, and in AC, RS, and BS is 500,000. RS takes 2000 restarts.

The Figures 4.6 and 4.7 show that the solutions of BS and RS are much closer to the solutions of RB-AC than those AC produced. The Figures 4.8 and 4.9 show that BS has a similar profile of time complexity as RB-AC has, and that the time complexities of RS are much lower than those of BS. Note that the empty places in the figures correspond with problem instances that are unsolvable.

Furthermore, in solving CSP $(40, 20, p_1, p_2)$, we tested the approximate algorithm RS for a variety of parameter combinations; the parameters are the number of restarts, the value-selection ordering, the upper bound on the number of backtracks, and the search-depth adaptation. In the experiments, p_1 takes the values 0.3 and 0.5. These values of p_1 are examples of CSP with a low and high number of constraints, respectively. p_2 takes the values 0.7 and 0.8. These two values of p_2 represent the loose and tight constraints in CSPs, respectively. Note that when p_2 is 0.6, the CSPs are already over-constrained. The experimental results for CSPs

Figure 4.6: CSP(20,10,0.3,p2)

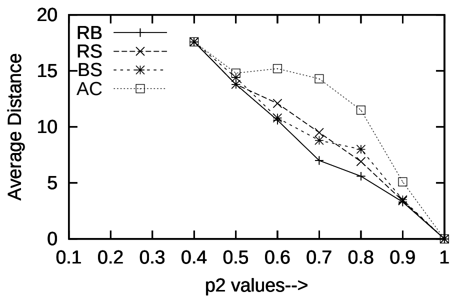


Figure 4.7: CSP(20,10,0.7,p2)

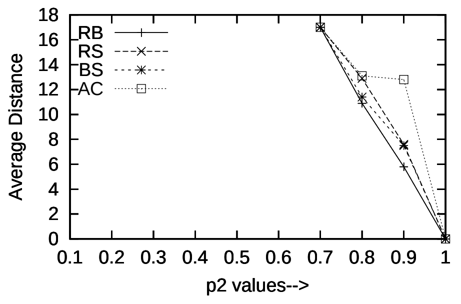


Figure 4.8: CSP(20,10,0.3,p2)

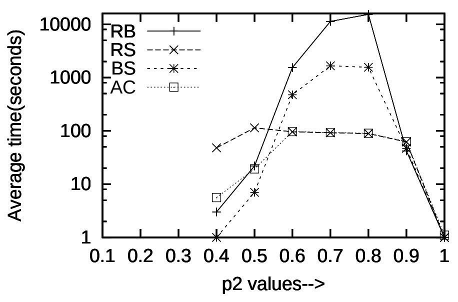
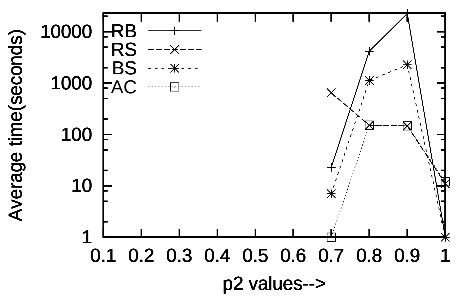


Figure 4.9: CSP(20,10,0.7,p2)



$(40, 20, 0.3, 0.7)$ and $(40, 20, 0.3, 0.8)$ are presented by the Figures 4.10 to 4.17. In these Figures, the number of restarts takes values from 100 to 7000. The marked points on the curves represent the average distances and average time consumption produced by the indicated approach over 10 CSP instances. One approach is compared with another in terms of the execution efficiency (time consumption) and the solution optimality (shortest distance) after a certain time elapse. Due to the similarity, the experimental results for p_1:=0.5 are not presented.

When p_2 is 0.7, we first compare the effects of the two approaches that have a distinct number of backtracks, but both of them are equipped with random-value selection (ra) and no-search-depth adaptation (NA). Figure 4.10 shows that a moderate number (4h:=400) of backtracks (ra4hNA) outperforms a large number (ub:=500,000) of backtracks (raubNA). Second, the previously better approach (ra4hNA) is taken to be compared with an approach (ra4hA) that applies search-depth adaptation while other parameters remain the same. Figure 4.12 shows that search-depth adaptation (ra4hA) outperforms the no-search-depth adaptation (ra4hNA). Third, again the previously better approach (ra4hA) is taken to be compared with an approach (mi4hA) that applies the mini-conflict value-selection heuristics while other parameters remain the same. Figure 4.14 shows that the mini-conflict value-selection heuristic (mi4hA) performs roughly equal to the random-value selection (ra4hA).

When p_2 is 0.8, we compare different approaches in a similar way as that p_2 is 0.7. Figure 4.11 shows that a moderate number of backtracks (ra4hNA) outperforms a large number of backtracks (raubNA) after 400 seconds. Between 50 and 400 seconds the two approaches perform roughly the same. Here, both approaches take the random-value selection and no-search-depth adaptation. Figure 4.13 shows that no-search-depth adaptation (ra4hNA) outperforms search-depth adaptation (ra4hA), while both of them take 400 backtracks and the random-value selection. Figure 4.15 shows that the mini-conflict value-selection heuristic with search-depth adaptation (mi4hA) outperforms the random-value selection with no-search-depth adaptation (ra4hNA), while both of them take 400 backtracks.

Now, we can figure out the proper parameter combinations in RS to solve the CSPs with different type of constraints. The Figures 4.10, 4.12 and 4.14 show that using a moderate number of backtracks and search-depth adaptation is crucial for solving the CSPs with tighter constraints. The value-selection heuristic is, however, not important since the mini-conflict heuristic does not definitely outperform the random value selection and vice versa. Figures 4.11, 4.13 and 4.15 show that the mini-conflict value-selection heuristic and a moderate number of backtracks with search-depth adaptation, denoted as MIMOA, outperforms other parameter combinations for solving the CSPs with looser constraints. Since it is usually difficult to determine whether a given CSP has looser or tighter constraints in practice, considering Figure 4.14 and Figure 4.15 together, MIMOA can be taken as the best parameter combination for solving general CSPs.

If we change one of the parameters in MIMOA, such as replacing the search-depth adaptation (mi4hA) with no-search-depth adaptation (mi4hNA), the quality of the produced solution is not better than that of MIMOA. Figure 4.16 shows that the search-depth adaptation certainly outperforms the no-search-depth adaptation for the CSPs with tighter constraints. Figure 4.17 shows that there is no preference for search-depth adaptation or no-search-depth adaptation in solving the CSPs with looser constraints. For the same reason in practice, MIMOA can still be viewed as the preferred parameter combination for solving general CSPs. Other parameter combinations have also been examined in the experiments. They did not outperform the parameter combination in MIMOA on the whole.

4.6 Chapter conclusions

In this chapter, we have proposed three repair-based methods for solving DCSPs, which include a complete repair-based algorithm (RB-AC) and two approximate algorithms (BS and RS). All algorithms aim at finding a solution for a CSP requiring a minimal or a near-minimal number of assignment changes with respect to its infringed solution. The complete algorithm applies an iterative deepening form of local search combined with constraint propagation. BS and RS give up the completeness by limiting the amount of search.

Through empirical studies, we conclude that the approximate algorithm RS with appropriate parameter combinations can be used to solve DCSPs as efficiently as possible for achieving a solution-maintenance objective, i.e., obtaining a solution with a near-minimal number of assignment changes. Hence, this chapter adequately addressed the first part of the general research problem given in the problem statement (Section 1.2).

Chapter 5

Repair-Based Scheduling

This chapter presents a study of Repair-Based Scheduling. In Section 5.1, the motivation for conducting a *repair* for an existing schedule is clarified. In Section 5.2, the following four issues are discussed: (1) why an innovative Repair-Based Scheduling approach is needed? (2) what kind of model modification should be made to the problem? (3) what objective should be achieved in Repair-Based Scheduling? and (4) what minimal perturbation function is appropriate for describing the objective? In Section 5.3, a thorough analysis for an unexpected event (e.g., a machine breakdown) that happens in a Job Shop is made. A *key*-operation on the breakdown machine is identified. The operations that need to participate in the repair are determined. Section 5.4 gives an overview of a novel Repair-Based Scheduling algorithm (denoted as RBS) for handling such unexpected events. The main ideas and techniques of the algorithm are illustrated. The pseudo codes of the RBS are given in Section 5.5. Section 5.6 presents and illustrates innovative heuristic procedures which are the result of different design choices of RBS. Finally, in Section 5.7, the routine of a new constraint-propagation technique is presented which determines the operations' start-time domains and prunes the search space adequately.

5.1 Motivation for repair-based scheduling

In section 3.7, we briefly introduced the reactive-scheduling concepts and corresponding methodologies. We saw that the scheduling requirements in a reactive scheduling phase are more than those in a predictive scheduling phase. In a real-world reactive-scheduling environment, the additional requirements (constraints) often include that the new schedule must differ minimally from the original schedule. The reason is that the original schedule represents an investment in planned resources, i.e., an allocation of machines and people. By executing the schedule a large number of interdependent processes has been set in motion. If an unexpected event occurs, keeping as much as possible the continuity of those processes is of vital importance. Hence, not disturbing the original schedule more than necessary is a key factor for reaching an agreement among all affected parties regarding the changes on the original schedule. In a common sense, minimizing the changes on the original schedule to obtain a new schedule is the right objective of repairing a schedule.

In the realm of reactive scheduling we focus in this thesis on Repair-Based Scheduling to be interpreted as generating a new schedule that has a minimal difference with the original schedule. The difference between new and original schedules may be explicitly expressed by some measures or functions. Although the reactive-scheduling systems currently in existence [?, ?] claim to emphasize on keeping the changes to the original schedule as minimal as possible, most of them primarily try to balance this objective with the traditional optimization objectives, such as minimizing the make-span, minimizing the mean tardiness of jobs etc. As pointed out in [?], there is a strong commercial demand for techniques that can better deal with the reactive scheduling problems to achieve the objective of Repair-Based Scheduling.

In Chapter 3, we outlined a number of powerful predictive (pre-established) scheduling methods based on constraint-directed search and introduced some reactive (repair-based) scheduling approaches. The methods developed in this chapter are also based on constraint-directed search. However, the focus of this chapter is on repairing a Job Shop Schedule. Thus, the notations and definitions associated with the JSSPs in Chapter 3 are still valid in this Chapter. In Chapter 3, we considered only the standard JSSPs in which (1) the processing times of operations are fixed values, (2) each machine can process only one operation, and (3) each operation can be processed by only one machine at a time. These three issues will be investigated more closely below.

During the execution of a schedule in a job shop, some unexpected events may happen (such as a machine breakdown). As a consequence, users may want to add new requirements. The result may be that the original JSSP model has to be changed to a new JSSP model. The original schedule of the original JSSP needs to be repaired (revised) to cope with the dynamic changes on the JSSP model. We are only interested in those dynamic JSSPs in which new requirements (constraints) are added. Since in these cases the original schedule may not be consistent with the newly added constraints, some modifications and adjustments must be made on the original schedule. For those dynamic JSSPs with constraint removals only, the original schedule of the original JSSP is still valid without need of modification and the new JSSP becomes simpler than the original one. We are not going to discuss the latter cases in this thesis. As we mentioned in the beginning of this paragraph, a resource shortage or failure, viz. a machine out-of-service (breakdown), is a typically unexpected event that may happen in a job shop. In the Chapters 5 and 6, we restrict our study on handling this kind of unexpected events in a job shop.

In the sections below, we discuss the strategies of repairing an existing schedule. Then, we present an innovative Repair-Based Scheduling strategy that is able to generate a new schedule with a near-minimal difference to the original schedule.

5.2 Repairing an existing schedule

Since a machine failure often introduces a schedule inconsistency, the whole or a part of the job-shop production cannot be implemented in accordance with the original schedule established for the original JSSP. In this case, there are roughly two reactive options available to industrial companies: (1) radical changes in the production system or (2) considerable improvement of existing systems. The improvement option provides a rich context of Repair-Based Scheduling methods. In order to satisfy the requirements of improvement, Repair-Based Scheduling must achieve two objectives simultaneously:

1. restore the feasibility of a schedule (now known to be infeasible because of the introduction of new conflicting constraints);
2. produce a new schedule which has a minimum difference with the original schedule.

To meet the above two objectives which are obviously different from those in predictive scheduling, repair action usually starts with identifying a set of problems [**?**, **?**, **?**] in the original schedule through localized schedule analysis as we will do in Section 5.3. Then, the repair activity proceeds with making local changes to minimize the potential (global) ripple impact and press optimization needs. Given the tightly coupled nature of scheduling decisions made in the original schedule, changes (e.g., a local conflict-resolving action) to one portion of the schedule often have ripple (non-local) effect. However, as pointed out in [**?**], it is generally not possible to predict the ripple effect of a change nor to bound the scope of change required to the original schedule in advance. Heuristic guidance may minimize this phenomenon, but problem combinatorics prevent its elimination. Thus, when reasoning about possible repair actions, an appropriate balance between achieving various scheduling objectives and being computationally efficient is a difficult task.

5.2.1 A new approach is needed

Once an original schedule needs to be repaired, the human schedulers often tend to adopt myopic “fire-fighting” tactics (where extinguishing one fire ignites the next). Such simple tactics keep the execution moving, but the global system behavior mostly deteriorates rapidly. As matters stand now, efficient search algorithms for schedule optimization (except for a very limited set of simple objectives such as make-span) are still not available and the amount of computation required for finding a solution is generally unpredictable. Consequently, constructing a cheap but suboptimal schedule that is then incrementally evolving to meet optimization objectives are well studied in repair strategies [?, ?]. The advantage of incremental repair is that one can interrupt the repair (evolving) process and use the interim result for execution when no more time is allowed for further repair. This means that a scheduling system can quickly respond to the unexpected environmental changes. Nevertheless, the available incremental repair methods focus on the most conflicting constraints (most contention resource) and start to repair the operations involved in these constraints (resolve the contention on the resource). Attendance to primary resource contentions can lead to the emergence of secondary resource contentions; and in some cases there is no dominant locus of resource contention in the overall manufacturing system. Moreover, even though an operation is not involved in the most conflicting constraint (i.e., the operation does not explicitly require a "repair" per se), it might be possible to create a better schedule by altering its original start-time assignment. For example, placing an operation uninvolved in the identified constraint much later in the schedule might cause little ripple effects and allow many operations (which are involved in the identified constraint) to fit in their place. This opportunity is missed if the uninvolved operations are not considered in the earlier repairing process. Therefore, how similar should the new version of the schedule be in relation to the old one is still an open question. Each of these circumstances suggests that different problem-structuring decisions, and dynamically-determined schedule-building strategies are needed.

Below, we present an innovative Repair-Based Scheduling approach which uses a completely different problem solving structure and accordingly appropriate heuristics to select operations to be repaired. In brief, the repair does not need to select the operations involved in the most conflicting constraint prior to other operations.

5.2.2 Model modification

Due to a machine breakdown, the start times of some unprocessed operations on the breakdown machine may need to be postponed. If no operation's start time need to be postponed which may happen in some scarce cases, the original schedule is still valid which can be kept to run without any change. However, in most cases of a machine breakdown, relaxing the due date (or make-span) to allow delays of the start times of some operations is necessary.

From a JSSP point of view, some parameters in the original JSSP *model* need to change when a machine breakdown takes place. Generally speaking, model modification reformulates the problem by constraint relaxation and (or) addition. The constraint relaxation in model modification includes throwing away already executed operations, changing release or due dates, increasing machine capacity, etc. It facilitates the solution of the problem. But the constraint relaxation is costly to implement in practice (e.g., buy new equipment to increase capacity, pay a fine for the production delay, sub-contract jobs to outside contractors). Thus, it should be made to the minimum extent possible. For this reason, we only assume two inevitable constraint relaxations in a model modification for Repair-Based Scheduling. One is relaxing the due date. Another is lessening the number of operations participating in the repair. The reason for the first relaxation has been stated in the first paragraph of this subsection. The latter relaxation is based on the fact that the executed portion of the original schedule no longer need to be considered with time elapses.

Besides the two above-mentioned general constraint relaxations in model modification, the constraint additions in the original JSSP model should be dealt with individually for each machine-breakdown instance. The added constraints in machine-breakdown events impose that some operations processed by a breakdown machine must be delayed with respect to their originally scheduled time and that the original schedule should be preserved as much as possible. When the execution of an operation is interrupted by a machine breakdown, a unary constraint must be added to the new JSSP model which imposes that the start time of the uncompleted part of the interrupted operation should be equal to the machine recovery time. The process time of the interrupted operation is changed to be the process time of the uncompleted part of the interrupted operation.

From a CSP point of view, Repair-Based Scheduling faces an extra set of constraints that must be resolved in the repair procedures. However, the existence of a feasible solution in Repair-Based Scheduling is guaranteed in the special modification context of assuming a constraint is “infinitely relaxable” along one dimension, i.e., operations can always be delayed. This flexibility can be counterbalanced by the inclusion of corresponding repair objectives to minimize relaxation to the extent possible (e.g., minimized the disturbance on the original schedule).

5.2.3 Objectives and criteria for repairing a schedule

Some relatively simple optimization functions can be found in the literature for predictive and reactive scheduling, such as the minimization of make-span [?], minimization of the average (or maximum) tardiness of operations (how much time the operations finish after their due date), or some combination of other attributes (for example, minimizing work-in-process combined with tardiness [?]).

For practical scheduling problems, it is desirable that multiple optimization objectives are satisfied simultaneously. However, optimization objectives often interact and conflict with one another. To optimize along one objective alone could jeopardize optimality along other objectives. The evaluation criteria for judging the acceptability of the outcome of a repair action are often multiple, conflicting, and context dependent. Therefore, it is difficult to describe the evaluation criteria and the associated trade-offs in a simple manner.

From Repair-Based Scheduling point of view, the original schedule should not be disturbed any more than necessary. So, *the fundamental tasks of schedule repair are to resume the feasibility and obtain a new schedule that has the minimum difference with the original schedule.* The latter requirement (objective) can be achieved by either minimizing the number of start-time changes of all operations or minimizing the sum of the new start-time delays of all operations with respect to their original start times. Two other possible objectives are minimizing the make-span and the largest delay of the start times of all operations. The four objectives mentioned above are formalized in Definition 21.

Definition 21 *Let the set V_1 denote all operations that participate in the repair, $Og : \Omega \to [0, H)$ be a original schedule of a JSSP, $ST : V_1 \to [0, H)$ be a new schedule of the operations in V_1. For any operation $o \in V_1$, the inequality $ST(o) \geq Og(o)$ is assumed to hold; the delay of its $ST(o)$ with respect to its $Og(o)$ is defined by $ST(o) - Og(o)$. The maximum completion time of all operations in the new schedule (make-span, MP), the sum of the delays of ST (SD), the largest delay of ST (LD), and the number of assignment changes with respect to the original schedule (NC) are defined by:*

$$MP(V_1) = \max_{o \in V_1}(ST(o) + p(o)) \tag{5.1}$$

$$SD(V_1) = \sum_{o \in V_1}(ST(o) - Og(o)) \tag{5.2}$$

$$LD(V_1) = \max_{o \in V_1}(ST(o) - Og(o)) \tag{5.3}$$

$$NC(V_1) = |\{o \in V_1, ST(o) \neq Og(o)\}| \tag{5.4}$$

These four objectives cannot be satisfied simultaneously, as it is illustrated by the following example (see Figure 5.1, too).

Example The box A of Figure 5.1 depicts an instance of a JSSP and its original schedule, where J_1 consists of three operations o_1, o_3 *and* $o_5 : o_1 \prec o_3 \prec o_5$; J_2 consists of two operations o_2 *and* $o_4 : o_2 \prec o_4$; o_1 is processed by M_1; o_2 *and* o_3 are processed by M_2; o_4 *and* o_5 are processed by M_3; $p(o_1) = 6,\ p(o_3) = 4,\ p(o_5) = 3; p(o_2) = 2,\ p(o_4) = 1$. The original schedule of the JSSP instance is: $Og(o_1) = 0,\ Og(o_2) = 1,\ Og(o_3) = 6,\ Og(o_4) = 3,\ Og(o_5) = 10$. The make-span of this schedule is 13. Let us assume that machine M_2 breaks down at time $t = 1$ and recovers at time $t' = 6$. The response time should be in the interval $[1, 6]$. In this particular case, the execution of o_2 has to be delayed at least 6 time units due to the period between the M_2 breaking down and recovering. However, the operation o_1 can be executed without being affected by the M_2 breaking down and the delay of o_2. This is because o_1 and o_2 are processed by a different machine, M_1 and M_2 respectively, and they do not belong to the same job. Therefore, they are not subjected to any resource (capacity) constraint or precedence constraint. Obviously, any start-time change (delay) of o_1 will increase the sum of the start-time delays. So, it is safe to exclude o_1 for repairing. The operations that have to participate

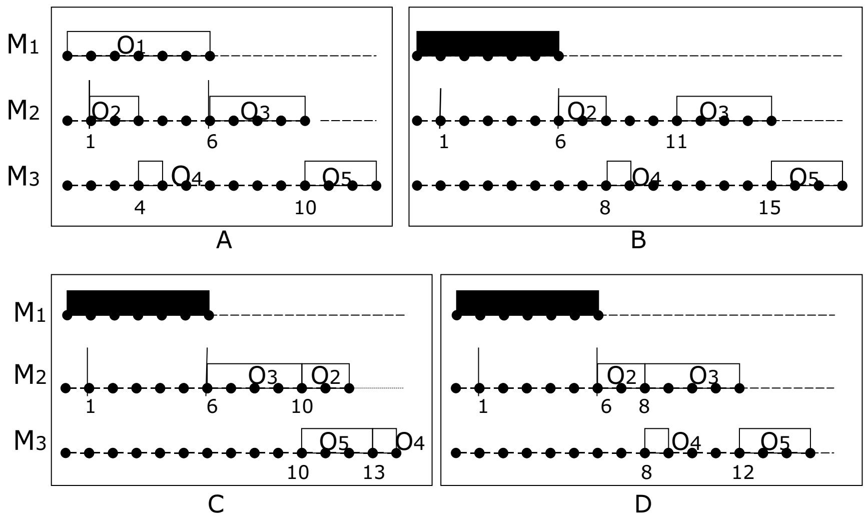


Figure 5.1: Example of reactive scheduling

in the repair are collected in the set V_1 ($V_1 := \{o_2, o_3, o_4, o_5\}$). Since o_2 and o_3 are on the breakdown machine, their earliest possible start time must be adjusted to 6. Consequently, the earliest possible start times of o_4 and o_5 must also be adjusted to new time points since o_2 and o_3 precedes o_4 and o_5, respectively.

1. The box B of Figure 5.1 depicts a *Right Shift* (denoted as RSH) strategy to get a new schedule in responding to the machine-breakdown event. This strategy implements a considerably less sophisticated reactive method that resolves conflicts by simply “pushing” the scheduled start times of all the operations in the set V_1 forward with 5 time units, which are determined by the o_2 on the breakdown machine. In this way, the new schedule can be obtained immediately after a machine breakdown. The execution of these designated shifts introduces neither time conflicts nor capacity conflicts, but the obtained schedule obviously is sub-optimal in terms of the make-span and the minimal difference with the original schedule. The value of MP, SD, LD and NC of the new schedule is 18, 20, 5 and 4, respectively.

2. The box C of Figure 5.1 depicts a strategy that minimizes the make-span (denoted as SMP) after M_2 breakdown. In this strategy, the start times of o_3 and o_5 are unchanged while the start times of o_2 and o_4 are pushed forward some time units. The value of MP, SD, LD and NC of the new schedule is 14, 19, 10 and 2, respectively.

3. The box D of Figure 5.1 depicts a strategy that tries to minimize the sum of

the new start-time delays of all operations in the set V_1. The value of MP, SD, LD and NC of the new schedule is 15, 14, 5 and 4, respectively.

From the Example and Figure 5.1, we can see that schedule C has the minimal make-span among three reactive schedules B, C and D. However, its SD is bigger than that of schedule D. As a matter of fact, minimizing make-span does neither automatically minimize the sum of the new start-time delays nor minimize the number of the start-time changes. Therefore, the objectives that minimize the difference between the original schedule and the new schedules must be dealt with in a way different from the traditional scheduling objectives, such as make-span.

5.2.4 Minimal perturbation problem

Since we modeled a JSSP as a CSP, the model modification after a machine breakdown actually creates a new CSP. Obtaining a feasible schedule and preserving the original schedule as much as possible can be viewed as finding a new schedule with a minimal *cost* λ (see definition 20).

According to Definition 20, the *cost function* λ is interpreted as the number of different start-time assignments between the new and old schedules, i.e., $\lambda = NC$. The algorithm developed for a DCSP in Chapter 4 could be used to minimize NC for a repair-needed JSSP. Although NC can be a measure in some application domains, it may not be suitable for Repair-Based Scheduling where a new schedule is required to have the minimum difference with respect to the original schedule. This is because keeping one operation's start time unchanged may force a big delay of another operation's start time. The big start-time delay of a single operation may be greater than the sum of the start-time delays of two or more operations where each operation make a smaller delay. Apparently, the latter has a smaller difference (closer start times) in total with respect to the original schedule. As shown in the Example of Figure 5.1, only minimizing the number of different assignments does not mean that operations have the new start times that are in total closer to the original start times. In order to measure the difference adequately (preferably evenly) and keep the operation's start times in total as close as possible to the original schedule, we define the *cost function* λ of a new schedule ST as the sum of its start-time delays as compared to the original schedule Og (the SD in Definition 21), i.e.,

$$\lambda(ST) := \sum_{o \in V_1} (ST(o) - Og(o)) = SD \tag{5.5}$$

Here, the new start times of all operations which participate in the repair are greater than or equal to their start times in the original schedule (i.e., $\forall o \in V_1, ST(o) \geq Og(o)$).

In this section, Repair-Based Scheduling is interpreted as generating a new schedule that has the minimal difference with the original schedule. Since the *cost function* λ (see formula 5.5) clearly and evenly represents the minimal difference between a new and an old schedule, we consider it as the core optimization function to be achieved in Repair-Based Scheduling activities.

Previously, the problem for obtaining a minimally disrupted schedule was formally called *Minimal Perturbation Problem* by Sakkout and Wallace [?]. Although their perturbation function differs from our cost function λ, we have adopted the name minimal perturbation problem, where Sakkout and Wallace also measure their perturbation function in terms of costs. An elaboration of their function shows similarities to our function $\lambda(ST)$ in 5.5.

5.3 Further analysis of the problem

From a constraint-based scheduling perspective, Repair-Based Scheduling has to deal with an extra set of constraints added to the new JSSP model except for the precedence and capacity constraints in the original JSSP which must be satisfied in predictive scheduling. These new constraints are related to imposing start-time delays of some operations and insuring smaller differences as much as possible between the original schedule and the new schedules.

Generally speaking, schedule repair (revision) is often driven by detecting and analyzing constraint conflicts in the original schedule that are introduced by changes to the JSSP model. In Repair-Based Scheduling, the original JSSP together with the original schedule needs to be scrutinized so that the proper decision can be made to cope with the unexpected events.

In this section, we first analyze the problem characteristics after an unexpected event (machine breakdown) has happened on a job-shop floor. Then, we identify some key issues which bring about the constraint changes (additions) to a JSSP model. In Subsection 5.3.1, a *key*-operation on the breakdown machine is identified. The operations that need to participate in the repair are determined in Subsection 5.3.2. The formal definition of a repair-needed JSSP is given in Subsection 5.3.3.

5.3.1 The key-operation

Given an instance of JSSP $(\mathcal{J}, \Omega, \mathcal{M}, H, \prec, J, M, p)$ and a function $Og : \Omega \to [0, H)$ representing an original schedule of the given JSSP, Figure 5.2 graphically represents a portion of an original schedule for a hypothetical job shop. For some machines, the start and finish times of some operations are shown in the Figure. In order to analyze conveniently the characteristics and complexities of the schedule-repair task, we assume that machine $M_i (M_i \in \mathcal{M})$ breaks down at time t; the anticipated recovery time period of M_i is r (we assumed that the anticipated recovery time period is always available in Repair-Based Scheduling); the operations processed by M_i is in the set $\Omega_i = \{o \mid o \in \Omega, M(o) = M_i\}$; the operations that are scheduled to execute on M_i in the interval $[t, t + r)$ are in the set:

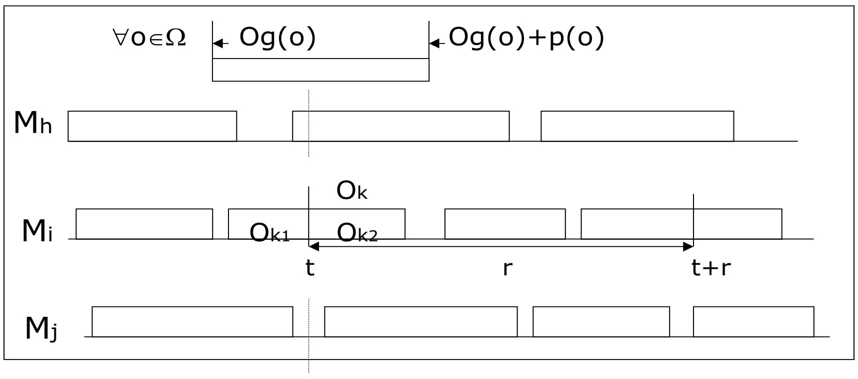


Figure 5.2: Machine M_i breaks down at time t.

$$\delta = \{o \mid o \in \Omega_i, Og(o) < t + r \wedge Og(o) + p(o) > t\}. \tag{5.6}$$

With the previous assumptions and corresponding notations, we proceed with our investigation and analysis of current problem structure by focusing on the breakdown machine. Two cases resulting from the event can be identified as follows.

1. $\delta = \emptyset$. This means that no operations are executing on M_i in the interval $[t, t + r)$, i.e., all operations processed by M_i either finished before time t or did not start yet at time $t + r$ according to the original schedule. Thereby, rescheduling or repairing the original schedule is not necessary. The original schedule is still valid.

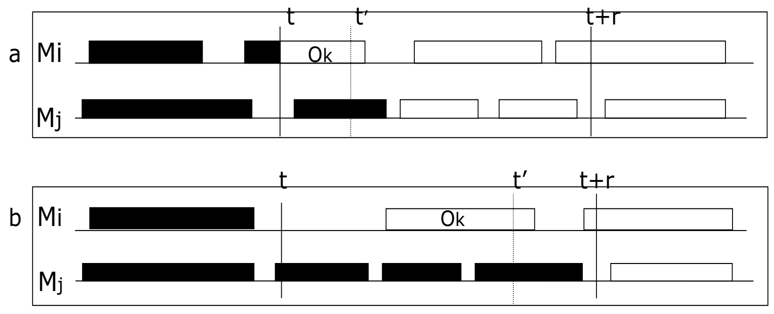
2. $\delta \neq \emptyset$. This means that at least one operation cannot be processed according to the original schedule. Thus, reactive scheduling is required.

Hereafter, we concentrate our analysis on the machine breakdown instances in which the set δ is not empty.

In order to take proper action, we first identify the *key-operation* o_k which has the minimal start time in the set δ, i.e.,

$$o_k \in \delta \ (\forall o \in \delta,\ Og(o_k) \leq Og(o)) \tag{5.7}$$

Obviously, the operations of which start times are greater than that of o_k on M_i may need to be adjusted. In case of $Og(o_k) \geq t$, the new constraints, which specify that the earliest possible start times of operations in δ must be greater than or equal to the recovery time of the breakdown machine, should be added to the new JSSP model.



where M_i breaks down at t and recovers at $t + r$;

Figure 5.3: Two cases, the position of O_k is different in both cases.

In case of $Og(o_k) < t$, the execution of o_k is interrupted by the M_i breakdown. Thus, o_k can be split into two operation sequences (o_{k1} o_{k2}) (as illustrated in Figure 5.2) reflecting already completed and uncompleted portions of o_k. We pointed out previously that only non-preemptive JSSPs in which an operation cannot be interrupted are considered in Repair-Based Scheduling. For the interrupted execution of o_k, which is enforced by the M_i breakdown, we try to reduce the interruption as much as possible by immediately restoring the execution of o_{k2} after machine M_i has recovered. So, an unary constraint that specifies the start time of o_k (i.e., o_{k2}) which should be $t + r$, is added to the new JSSP model. The process time of o_k is changed into the process time of o_{k2} which is shown in Equation 5.8.

$$p_{new}(o_k) = p(o_k) - (t - Og(o_k)) \tag{5.8}$$

The other constraints that should be added to the new JSSP model are as same as those in the case of $Og(o_k) \geq t$.

5.3.2 Operations that participate in the repair

Determining the operations that must participate in the repair process is an important issue. The set of repair-needed operations V_1 should only contain operations of which the original start time might have to change in order to minimize the λ value specified by the objective function. Obviously, V_1 will contain all operation that can no longer be assigned their original start times as well as all their successors. Hence, we only need to determine whether operations that precede the operations that might no longer be assigned their original start times, should also participate in the repair process.

According to the original start and finish time of the *key* operation, all operations of the given JSSP can be classified into three sets. The first set contains the operations of which the original start time is greater than the original finish time of the *key* operation. These operations might be affected by the machine-breakdown event and must participate in the repair activities. We use a set V_1 to collect all operations in this set and the *key* operation.

$$V_1 = \{o_k\} \cup \{o \in \Omega \mid Og(o) \geq Og(o_k) + p(o_k)\}$$

The second set contains the operations of which the original finish time is less than the original start time of the *key* operation. This means that any possible changes caused by the breakdown machine happen after these operations finishing their execution. Hence, the operations in this set can be excluded from the repair activities.

The third set contains the operations (processed by machines other than the breakdown machine) of which the original start time is less than and the original finish time is greater than, respectively, the original finish time and the original start time of the *key* operation. The following proposition will be used to determine whether additional operations in this set must be considered.

Proposition 5 *Let $V_1 = \{o \in \Omega \mid Og(o) \geq Og(o_k) + p(o_k)\} \cup \{o_k\}$ be a set of operations that participate in the repair process. The start times of these operations may be delayed in the repair process. Operations that are not in V_1 receive their original start times.*

Adding any operation $o'(o' \notin V_1)$ to V_1 will not decrease the λ value of the objective function (Equation 5.5) on the set V_1.

Proof

Let MSD be the minimal sum of the start-time delays on the set V_1.

Then there exists a complete assignment ST_1 repairing the operations in V_1 such that:

$$MSD = SD_{ST_1}(V_1) = \min_{ST}\left(\sum_{o \in V_1}(ST(o) - Og(o))\right) = \sum_{o \in V_1}(ST_1(o) - Og(o)).$$

Let $V_1' = V_1 \cup \{o'\}$ $(o' \notin V_1)$. This is the set of operations that are allowed to be delayed. Moreover, let ST_2 be a complete assignment repairing the operations in V_1'. Clearly, if o' is not delayed, i.e. $ST_2(o') = Og(o')$, we have $SD_{ST_2}(V_1') = MSD$. So assume that o' is delayed, i.e. $ST_2(o') > Og(o')$.

There are two possibilities for this delay. We show that with both we run into a contradiction:

1. There is a predecessor o^* of o' that is delayed in ST_2. Clearly, for no $o \in V_1$ we have $o \prec o'$, since this would imply $o' \in V_1$. Hence, $o^* \notin V_1'$ and therefore, by the assumption concerning V_1', $ST_2(o^*) = Og(o^*)$; contradiction.

2. There is a $o^* \neq o'$, $M(o') = M(o^*)$ and o^* interferes with o' in ST_2. That is,

$$ST_2(o^*) + p(o^*) > Og(o')$$

   and

$$Og(o') + p(o') > ST_2(o^*).$$

   Clearly, $o^* \in V_1'$ since otherwise we would have $ST_2(o^*) = Og(o^*)$ and there would be no conflict between the schedules of o^* and o'. But then, since $o' \notin V_1$, it follows by definition of V_1 that $Og(o') < Og(o^*)$ and since both are scheduled on the same machine we have $Og(o') + p(o') < Og(o^*) < ST_2(o^*)$, contradicting $Og(o') + p(o') > ST_2(o^*)$.

Hence, our assumption that $ST_2(o') > Og(o')$ does not hold. Therefore, $ST_2(o') = Og(o')$ and $SD_{ST_2}(V_1') = MSD$. □

From Proposition 5, any operation which is not in the set V_1 can be excluded from the repair without influencing the minimal sum of the start-time delays of the repaired schedules on the operation set V_1 (the operations that participate in the repair).

By synthesizing the analysis in this section, the set V_1 which represent the operations that must participate in the Repair-Based Scheduling can be explicitly expressed as follows:

$$V_1 = \begin{cases} \{o_k\} \cup \{o \in \Omega \mid Og(o) \geq Og(o_k) + p(o_k)\} & \text{if } \delta \neq \emptyset \\ \emptyset & \text{if } \delta = \emptyset \end{cases} \tag{5.9}$$

Figure 5.3 (a,b) shows the operations (blank) which must participate in the repair and those operations (black) which do not need to participate in the repair.

Some other unexpected events, such as supply-delivery delays, have similar characteristics. Hence, they can be treated in a similar way.

5.3.3 Formal definition of a repair-needed JSSP

The problem to be solved by the Repair-Based Scheduling approach depends on the JSSP instance, the original schedule Og for the problem instance, and the time interval $[t_{failure}, t_{recovery})$ in which some machine $bdm \in \mathcal{M}$ is unavailable because of some unexpected event. We assume that the prediction of the machine recovery is always available. An instance of a repair-needed JSSP can be expressed by a tuple $(V_1, \mathcal{M}, \prec, Og, p, M, rt)$ where:

- V_1 (as shown in Equation 5.9) is the set of the operations to be considered in the repair process;
- $\mathcal{M}$ is a set of machines;

- $\prec \subseteq \Omega \times \Omega$ is a set of precedence constraints;
- $Og : \Omega \rightarrow [0, H)$ is the original schedule for the operations of a JSSP;
- $p : \Omega \rightarrow \mathbb{N}$ gives the processing time of each operation;
- $M : \Omega \rightarrow \mathcal{M}$ is a function specifying the machine requirement of an operation;
- $rt : V_1 \rightarrow \mathbb{N}$ is a function specifying the release time of an operation $o \in V_1$. Here:

  $rt(o) = max(Og(o), max(\{Og(o') + p(o') | M(o') = M(o), o' \in \Omega \wedge o' \notin V_1\}))$, if $M(o) \neq bdm$;

  $rt(o) = max(Og(o), t_{recovery})$, if $M(o) = bdm$ and $t_{failure} \leq Og(o_k)$;

  $rt(o) = max(Og(o), t_{recovery} + p_{new}(o_k))$, if $M(o) = bdm$ and $Og(o_k) < t_{failure} < Og(o_k) + p(o_k)$;

  where o_k is the key operation (specified by Equation 5.7) on the failure machine; $p_{new}(o_k)$ is given by Equation 5.8.

A 'repaired' schedule ST for the new JSSP is a complete start-time assignment of the operations in V_1, $ST : V_1 \rightarrow \mathbb{N}$ such that:

- the total delays to the original schedule $\sum_{o \in V_1}(ST(o) - Og(o))$ is minimal;
- for every $o \in V_1 : rt(o) \leq ST(o)$;
- for every $(o, o') \in \prec$: $ST(o) + p(o) \leq ST(o')$;
- for every $o, o' \in V_1$: if $M(o) = M(o')$, then either $ST(o) + p(o) \leq ST(o')$ or $ST(o') + p(o') \leq ST(o)$.

If a Repair-Based Scheduling system responds to the unexpected events at time $t^\star$, $t^\star$ must be in the interval $[t, t + r]$. That is to say, the Repair-Based Scheduling system must generate a new schedule before the time $t + r$. Therefore, except for achieving the minimal λ value of the perturbation function, the efficiency of the Repair-Based Scheduling system is an important issue to be considered when developing such a system.

In the next section, we first give an outline of a repair-based scheduling algorithm which is designed to solve the repair-needed JSSPs. Then we illustrate the details of a new operation selection heuristic and the constraint-propagation techniques used in the algorithm.

5.4 A repair-based scheduling algorithm (RBS)

To overcome the drawbacks of the incremental repair strategy, an innovative Repair-Based Scheduling algorithm (RBS) is developed. The RBS will be outlined and illustrated in this section.

5.4.1 Outline of the RBS

The RBS algorithm is developed to generate a new schedule for a JSSP after occurrence of a machine failure. In rescheduling all operations that must participate in the repair, RBS tries to minimize the perturbation value λ. Since minimizing λ is the optimization objective and a constraint-based approach is the fundamental of RBS, the optimization objective is approximated by iteratively putting a tighter constraint on the maximum allowed value for λ. Each time a valid solution is found, the λ is calculated and a new tighter constraint on the allowed λ value is introduced. In each iteration, RBS completely restarts search a number of times for generating different search paths. Each search node is coupled with a *semi-randomized* operation selection heuristic.

Besides the constraint-propagation techniques used in predictive scheduling (*Two Consistency Checking* and *Edge Finding*), a newly developed constraint-propagation technique which impose the minimal perturbation value λ to reduce the search space (start-time domains) is used in RBS. The technique deals with the really consumed sum of delays by already assigned operations and the delays which are anticipated to be consumed in future search processes by yet unassigned operations. Both of the delays are subtracted from the remaining total delays and enforced as an unary constraint on the unassigned operations' start-time domains. Since the constraint on the maximum allowed value of λ is a global constraint involving the start times of all operations, the pruning effect of the new constraint propagation is insufficient to solve instances in which the maximum value for λ becomes tight. Therefore, a new operation selection heuristic has been developed.

The heuristic is a *semi-randomized* operation-selection heuristic that constructs a schedule from left to right (assuming that operations are ordered from left to right with respect to the precedence constraint). This also implies that given a partial schedule the heuristic determines which operation will be the next operation on some machine. The heuristic consists of four steps. First, the heuristic determines the operations that are candidates for assigning a new start time at the current node of the search tree. Second, the heuristic randomly selects a machine. It does this by randomly selecting one of the candidate operations where each operation has an equal probability of being selected. Third, the heuristic estimates the ripple effect of each candidate operation that requires the selected machine. Here, the *ripple effect* is defined as the influence on the start-time domains of the unassigned operations when the operation under consideration is assigned its best start time in the current search node. The candidate operation with the lowest estimated ripple effect is chosen. The search is thus redirected and operation-selection decision is repaired. Fourth, the operation selected by the heuristic is assigned its best start time.

To sum up, *RBS* is a rescheduling algorithm which combines generating a new schedule with achieving the repair objective (minimizing the λ value of the perturbation function).

In the successive subsections, we illustrate the semi-randomized operation selection heuristic and the constraint-propagation techniques used in RBS.

5.4.2 Main ideas behind the RBS

The success of the constraint-directed search in predictive scheduling motivated us to adopt its framework as the basis of the novel RBS algorithm. Since Repair-Based Scheduling problems are NP-hard problems [**?**], a randomized-restart strategy which has been successfully used when generating a predictive schedule is adopted. However, minimizing the λ value of the perturbation function in RBS is completely different from that of minimizing the make-span. A more sophisticated operation-selection heuristic, which can map the optimization needs and opportunities implied by the specific search states to the current repair actions, is required. Moreover, a particular constraint-propagation technique which can impose the minimal perturbation requirement (λ value) and simultaneously reduces the search space is necessary.

In order to search for a solution, RBS has to select an operation at each search node, to assign a start time to the selected operation and to propagate the current assignment to the start-time domains of yet unassigned operations.

Since we wish to minimize the perturbation value λ, a new schedule must be left-justified. Left-justified schedules can be built in a way from left to right such that the branching factor in each search node can be significantly reduced. During the search processes, some operations must start before other operations and some operations can start before other operations without increasing the λ value. We use this knowledge to filter the operations that are candidates to be assigned in the current search node. Therefore, when choosing the operation to be assigned next, we only consider the operations that can start before the minimal earliest finish time (EFT) of all unscheduled operations. We denote this set of candidate operations by CV.

In order to make a good choice, we try to estimate the ripple effect of a candidate operation. However, we do not evaluate all operations in CV with respect to their ripple effects. Instead, we evaluate the operations in a subset of CV with respect to their ripple effects. First, an operation in the candidate set CV is randomly selected and the corresponding machine is determined. Here, every operation in CV has an equal probability of being chosen. So, the machine with higher contentions (the machine with more candidate operations on which processing could start) has a higher probability to be chosen. Second, all operations in CV that are processed by the chosen machine (a subset of CV) are evaluated with respect to the ripple

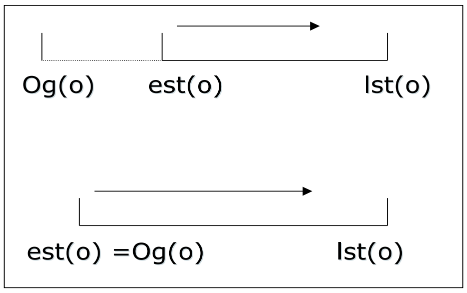


Figure 5.4: Operation start-time assignment.

effects. Third, the operation with the minimal estimated ripple effect is chosen to replace the randomly selected operation as the next one on the selected machine.

The main reasons for developing such semi-randomized heuristic can be summarized in two points. (1) Since the delays consumed by each assignment are obvious, we know exactly how much delay is consumed by the start times of already assigned operations. However, we do not know how much delay will be consumed by yet unasssigned operations. The heuristic should make an estimate of how much delay the start times of unassigned operations are to consume at least, and then lead the search process in the direction determined by the lowest estimated ripple effect. (2) In the set CV, no two operations belong to an identical job (the operations in a job are assigned start times from left to right). However, some operations in CV might be processed by the same machine. We know that on a machine, different orders of operations have different ripple effects. If there exists at a search node several candidate operations in CV to be processed by the same machine, the operation that is selected to be the next operation on that machine may have a significant impact on the ripple effects. Thus the operations in a subset of CV, which are processed by the same machine, are evaluated with respect to their ripple effects. The subset is randomly determined by selecting randomly an operation from CV.

We have also studied two alternative approaches. In the first alternative, the operation with the lowest estimated ripple effect in CV will be assigned next. In the second alternative, a machine is randomly selected with equal probabilities. Both alternatives will be dealt with in the next chapter.

After selecting an operation, RBS always assigns the operation the earliest possible start time since it constructs a left-justified schedule (see Figure 5.4).

Following an operation start-time assignment, the constraint propagation will be performed over the start-time domains of all unassigned operations. If the start-time domain of an unassigned operation becomes empty after the constraint propagation, then the currently selected operation cannot be the next operation on the selected machine. The operation must be scheduled after at least one of the other operations on that machine. So, the earliest possible start time of the operation should be

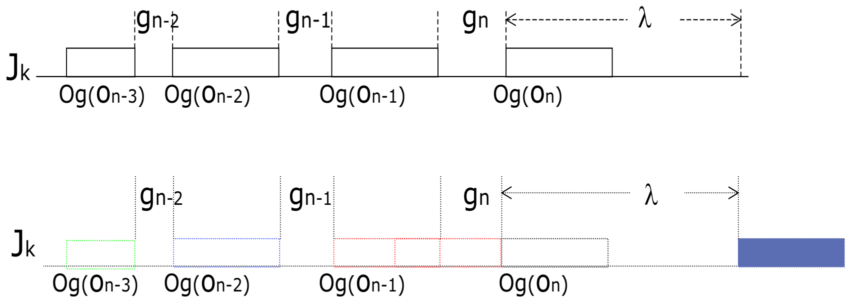


Figure 5.5: Start-time domain calculation.

postponed to $max(EFT(\{o'|o' \in V_x \wedge M(o') = M(o)\}), est(o) + 1)$ (where V_x is the set of operations which have not been assigned yet).

5.4.3 Propagating the λ value

As indicated in Section 3.6, search can drastically be reduced by enforcing various degrees of consistency which are implemented by constraint-propagation techniques. As well as in predictive scheduling, Repair-Based Scheduling needs to implement proper constraint propagations in balancing the complexity and the facilitating of the search effort. In this subsections, we introduce a new constraint-propagation technique which was first proposed in our paper [**?**]. The technique imposes an unary constraint derived from the λ value of the minimal perturbation function on the start-time domains of all unassigned operations.

Given the objective of Repair-Based Scheduling as established in Subsection 5.2.4, we can use this objective to prune the search space. That is, applying constraint propagation to eliminate impossible start times from an operation's start-time domain in which the latest possible start times are "infinitely" relaxed.

From the original schedule point of view, the Repair-Based Scheduling assigns each operation involved in the repair a new start time. According to Equation 5.5, λ is a measure to rate the difference between the original schedule and the repaired schedule. Let us assume that the operations of all but one job have the same start times as those in the original schedule and that one job is causing the maximum allowed delay λ. Figure 5.5 gives an illustration of the last four operations of such a job J_k. Note that the total order of the operations on job J_k is defined by the precedence constraint which will not change in Repair-Based Scheduling.

Let $Og(o_n)$, $Og(o_{n-1})$, $Og(o_{n-2})$ and $Og(o_{n-3})$ be the original start times of the last four operations in the original schedule; $g_i = Og(o_i) - (Og(o_{i-1}) + p(o_{i-1}))$ be the gap (slack) between the finish time of operation o_{i-1} and the start time of operation o_i. Given the maximum allowed delay λ for the operations on job J_k, we can calculate the latest possible start times of these operations. We start with calculating the latest possible start time of the last operation o_n on the job J_k.

Since o_n is the last operation of job J_k, by assuming that all other operations in the repair-needed JSSP will be assigned their original start times, the new start time of o_n can thus be delayed to the maximum λ value (not infinitely be delayed). So, the latest possible start time of o_n in the repaired schedule is bounded by:

$$lst(o_n) \leq Og(o_n) + \lambda$$

The maximum time period of the delay of operation o_{n-1} may be delayed depends on the gap g_n. If this g_n is larger than or equal to the maximum λ value, o_{n-1} can be delayed without considering o_n. Hence, the latest possible start time of o_{n-1} is bounded by $lst(o_{n-1}) \leq Og(o_{n-1}) + \lambda$. If g_n is smaller than the maximum λ value, o_{n-1} can be shifted until the gap is zero at first, and then both o_n and o_{n-1} are shifted. When shifting both o_n and o_{n-1}, a shift of one time unit causes an increase of two units of the λ value. Thus, the maximal remaining space for the delay of o_{n-1} start time is $\lfloor(\lambda - g_n)/2\rfloor$. As a result, the latest possible start time of o_{n-1} is bounded by:

$$lst(o_{n-1}) \leq \begin{cases} Og(o_{n-1}) + \lambda & \text{if } g_n \geq \lambda \\ Og(o_{n-1}) + g_n + \lfloor(\lambda - g_n)/2\rfloor & \text{if } g_n < \lambda \end{cases}$$

The latest possible start time of o_{n-2} is determined by g_n, g_{n-1} and λ. Three situations should be taken into account, i.e., (1) $g_{n-1} \geq \lambda$, (2) $g_{n-1} < \lambda \wedge 2g_n + g_{n-1} \geq \lambda$, and (3) $2g_n + g_{n-1} < \lambda$. The latest possible start time of o_{n-2} is bounded by:

$$lst(o_{n-2}) \leq \begin{cases} Og(o_{n-2}) + \lambda & \text{if } g_{n-1} \geq \lambda \\ Og(o_{n-2}) + g_{n-1} + \lfloor\frac{\lambda - g_{n-1}}{2}\rfloor & \text{if } g_{n-1} < \lambda \wedge wsg \geq \lambda \\ Og(o_{n-2}) + sg + \lfloor\frac{\lambda - wsg}{3}\rfloor & \text{if } wsg < \lambda \end{cases}$$

where: $sg := g_{n-1} + g_n$, $wsg := 2g_n + g_{n-1}$.

The latest possible start time of any operation o_{n-j} ($j \geq 3$) on job J_k thus can be bounded by the formula below.

$$lst(o_{n-j}) \leq \begin{cases} Og(o_{n-j}) + \lambda & \text{if } g_{n-j+1} \geq \lambda \\ Og(o_{n-j}) + sg_1 + \lfloor\frac{\lambda - wsg_1}{2}\rfloor & \text{if } wsg_1 < \lambda \wedge wsg_2 \geq \lambda \\ \vdots & \vdots \\ Og(o_{n-j}) + sg_j + \lfloor\frac{\lambda - wsg_j}{j+1}\rfloor & \text{if } wsg_j < \lambda \end{cases}$$

Where:

$$sg_x := \sum_{q=n-j+1}^{n-j+x} g_q$$
$$wsg_x := \sum_{q=n-x}^{n-1} (n-q) g_{2n-q-j}$$
$$1 \leq x \leq j.$$

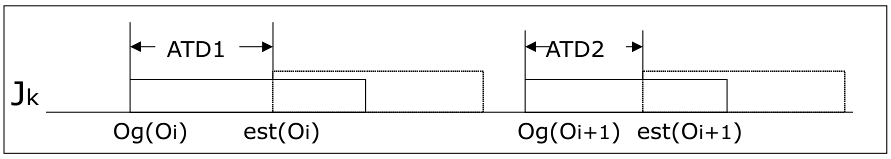


Figure 5.6: Anticipated delays.

The latest possible start times of operations on other jobs $J_i(i \neq k)$ can be calculated by the same formulas in which g_i depends on the gaps (slacks) between the operations on the job J_i.

5.4.4 Propagating allowed sum of delays

In Subsection 5.4.3 and our paper [?], a given λ value of the minimal perturbation function is assumed to be the maximum allowed sum of delays for all operations involved in repair. Propagating the λ value as described in Subsection 5.4.3 imposes a unary constraint on the operations' start-time domains (latest possible start times). However, two factors have not been taken into account in the previous subsection. Operations that have already been assigned a new start time may not have the same start time as in the original schedule. Moreover, the domain of an unassigned operation may no longer contain the start time of the original schedule (its earliest possible start time is delayed). In both cases, a proportion of the maximum λ value will certainly be used for these operations and can therefore no longer be used by an unassigned operation. Hence, instead of using the maximum λ value, we should use the remaining λ value AD (the maximum *Allowed Sum of Delays*) from which the delays that have been consumed and will certainly be consumed have been subtracted. Obviously at the current search node, propagating the AD as a unary constraint on all unassigned operations' start-time domains thus will further prune the search space.

For a convenient description, the delay of the earliest possible start time of an unassigned operation is called *anticipated delay* (ATD) of the operation. ATD1 and ATD2 in Figure 5.6 show this kind of delays for the unassigned operations on job J_k.

In the last subsection, an operation's latest possible start time is determined by setting the current λ value as the maximum allowed start-time delay for the operations on each job, where the gap (slack) between two adjacent operations on the same job is assumed to be a fixed value (see Figure 5.5), i.e., $g_i = Og(o_i) - (Og(o_{i-1}) + p(o_{i-1}))$ (if $o_{i-1} \prec o_i$). However, when propagating the AD from which the unassigned operations' ATDs have been subtracted, the gap (slack) between two adjacent operations on a job may also change. Figures 5.7, 5.8, 5.9 and 5.10 show four different situations in which the new gaps between two adjacent operations

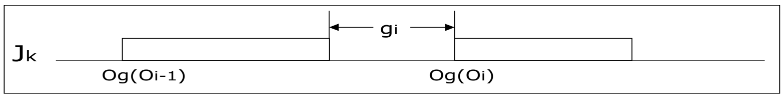


Figure 5.7: $est(o_i) = Og(o_i) \wedge est(o_{i-1}) = Og(o_{i-1})$.

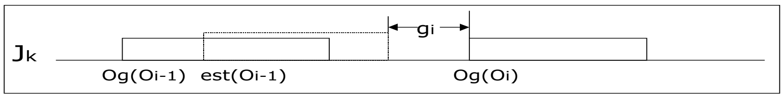


Figure 5.8: $est(o_i) = Og(o_i) \wedge est(o_{i-1}) > Og(o_{i-1})$.

should be calculated by the formulation below:

$$g_i = \; est(o_i) - (est(o_{i-1}) + p(o_{i-1}))$$

Moreover, from the analysis in Subsection 5.4.3, on how much delay could be shared by two adjacent operations, it was clear that it depends on the gap between them. Thus the maximum allowed delay for an operation's start time also depends on the gap between the operation and its successors on the same job. As a result, the formulations in the last subsection are no longer correct for calculating the operations' latest possible start times.

In order to propagate the AD as a unary constraint on all unassigned operations' start-time domains, we developed an algorithm to propagate the Allowed Sum of Delay. Let us assume that the start times of the operations on one job maximally have a delay of an allowed sum of AD from which the sum of ATDs and the delays made by already assigned operations have been subtracted. To get the latest possible start times of the operations on a job, similar steps as in the last subsection are adopted. We first calculate the latest possible start time of the last operation o_n on a job then calculate the other operations' latest possible start times.

If o_n is the last operation on job J_k, the maximum allowed delay of $ST(o_n)$ is AD but the start point of $ST(o_n)$'s delay is dependent on whether the current $est(o_n)$ is greater than $Og(o_n)$. If $est(o_n) - Og(o_n) > 0$ holds, the ATD $(est(o_n) - Og(o_n))$ has been subtracted from AD. So, when calculating the latest possible start time of o_n, the start point of $ST(o_n)$ delay becomes its earliest possible start time instead of its original start time. If $est(o_n) - Og(o_n) > 0$ does not holds, $est(o_n) = Og(o_n)$

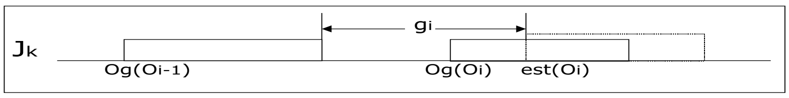


Figure 5.9: $est(o_i) > Og(o_i) \wedge est(o_{i-1}) = Og(o_{i-1})$.

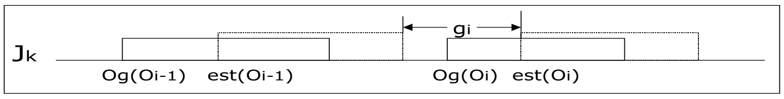


Figure 5.10: $est(o_i) > Og(o_i) \wedge est(o_{i-1}) > Og(o_{i-1})$.

holds. The start point of $ST(o_n)$ delay is $Og(o_n)$ which can be replaced by $est(o_n)$. Therefore, when o_n's ATD has been subtracted from AD, the start point of $ST(o_n)$ delay can be expressed by $est(o_n)$. The latest possible start time of o_n in the repaired schedule is bounded by:

$$lst(o_n) \leq \ est(o_n) + AD.$$

The time period that operation o_{n-1} may be delayed depends on the gap g_n and AD. For the same reason as o_n, the start point of $ST(o_{n-1})$ delay can also be expressed by $est(o_{n-1})$. If $g_n \geq AD$ holds, the maximum allowed delay of $ST(o_{n-1})$ is equal to $est(o_{n-1}) + AD$. If $g_n < AD$ holds, the maximum allowed delay of $ST(o_{n-1})$ depends on the delay of $ST(o_n)$. Once the $ST(o_{n-1})$ delays g_n time units, the remaining time units $AD - g_n$ should be shared by o_n and o_{n-1} start time delays. Thus, the maximum allowed delay for $ST(o_{n-1})$ is $est(o_{n-1}) + g_n + \lfloor (AD - g_n)/2 \rfloor$. Hence, the latest possible start time of o_{n-1} is bounded by:

$$lst(o_{n-1}) \leq \begin{cases} est(o_{n-1}) + AD & \text{if } g_n \geq AD \\ est(o_{n-1}) + g_n + \lfloor (AD - g_n)/2 \rfloor & \text{if } g_n < AD. \end{cases}$$

The latest possible start time of any operation o_{n-j} ($j \geq 2$) on job J_k thus can be calculated by the formula below.

$$lst(o_{n-j}) \leq \begin{cases} est(o_{n-j}) + AD & \text{if } wsg_1 \geq AD \\ est(o_{n-j}) + sg_1 + \lfloor \frac{AD - wsg_1}{2} \rfloor & \text{if } wsg_1 < AD \wedge wsg_2 \geq AD \\ \vdots & \vdots \\ est(o_{n-j}) + sg_j + \lfloor \frac{AD - wsg_j}{j+1} \rfloor & \text{if } wsg_j < AD \end{cases}$$

Where:

$$\begin{array}{l} sg_x := \sum_{q=n-j}^{n-j+x} g_q \\ wsg_x := \sum_{q=n-x}^{n-1} (n-q) g_{2n-q-j} \\ 1 \leq x \leq j. \end{array}$$

Based on the problem-solving exploration, a Repair-Based Scheduling (RBS) algorithm is developed. The pseudo codes of RBS are presented in Section 5.5.

5.5 Procedures in the RBS

Let $(\mathcal{J}, \Omega, \mathcal{M}, H, \prec, J, M, p)$ be an instance of a JSSP and $(V_1, \mathcal{M}, \prec, Og, p, M, rt)$ an instance of a repair-needed JSSP; AD denotes the maximal allowed sum of delays for the start times of all unassigned operations; $D(o)$ denotes the discrete domain of possible start times of the operation o. The pseudo codes of RBS are presented in the Figures 5.11, 5.12, 5.13 and 5.14. The RBS algorithm solves the Minimal Perturbation Problem by calling the procedure *Iterative_Improve*(V).

5.5.1 Iterative improvements and restarts

The procedure *Iterative_Improve* is given in Figure 5.11. It first identifies newly introduced constraints caused by the machine-breakdown event to the original schedule, such as the start times of some operations on the breakdown machine must be enlarged with an amount greater than or equal to the time of the machine recovery. Then, it determines a subset of V, V_1, which contains all operations that have to participate in the repair. As a result, a machine-breakdown instance of a JSSP is modeled as a new CSP which has explicit variable and constraint sets, whereas the variable domains (start-time domains of operations) are decided in each iteration independently. Moreover, based on the original schedule, the procedure *Iterative_Improve* generates a right-shift schedule through relaxing the completion-time bound but satisfying the newly added constraints, and the precedent and disjunctive constraints in the original JSSP. In more detail, the start times of all operations in the set V_1 are pushed forward with the same time units which are needed to recover the breakdown machine. This right-shift schedule is obviously a suboptimal schedule in coping with the new environment. However it can be obtained immediately after getting the predicted machine-recovery time. The sum of the start-time delays (λ value) of the right-shift schedule is used to be an initially allowed sum of delays (denoted as AD) for the start times of all operations in V_1 (i.e., $AD := \lambda$). Successively, two important parameters of the RBS are introduced. They are the backtracking factor BTF and the maximum number of restarts *nrRestarts*. The BTF gives the average number of chronological backtrack steps that is allowed for each operation, i.e., a maximum number of $\lfloor BTF * |V| \rfloor$ chronological backtrack steps is allowed before RBS restarts the search. With *nrRestarts* $= X$, RBS allows X number of tries to solve the instance, which corresponds to $X - 1$ times restarting the search. Inside the WHILE loop of the *Iterative_Improve*, a global variable *nrBT* is set to zero, which is used to account the number of chronological backtrack steps in one restart. Then a procedure *Solve* is called. If *Solve* returns $True$, the new schedule S' replaces the previous schedule S as the best solution and its λ value minus 1 is taken as the new AD for the next restart improvement. If *Solve* does not return $True$, *Solve* will be restarted with the same AD. Whether *Solve* returns $True$ or $False$, the restart number *nr* will increase by one. The iteration is repeated until the stop criterion ($nr > nrRestarts$) is reached, after which the algorithm

```
$Iterative\_Improve(V)$
   Identify new constraints introduced by machine breakdown;
   Create a subset $V_1$ of $V$;
   Generate a right shift schedule $S$;
   Initial $AD := \lambda(S)$;
   $maxBT := \lfloor BTF * |V| \rfloor$;
   $nr := 1$;
   while $(nr \leq nrRestarts)$
      $nrBT := 0$;
      if $(Solve(V_1, AD) = True)$
         $S := S'$;
         $AD := \lambda(S') - 1$;
      end if;
      nr:=nr+1;
   end while;
   return $S$;
end Procedure $Iterative\_Improvement$
```

Figure 5.11: Iterative Improvement.

returns the best schedule in terms of λ value. In each iteration, the cost bound AD is used to decide the start-time domains of operations and removing domain values that would lead to bigger λ values than those already found. Thus, each iteration, if a solution is found, guarantees that the λ value of a new schedule will be smaller than or equal to that of the previously found schedule.

5.5.2 Pre-treatment

The procedure *Solve* is given in Figure 5.12. At the beginning of the procedure *Solve*, three constraint-propagation procedures are activated, which includes two traditional constraint-propagation procedures used in predictive scheduling (Two Consistency Check and Edge Finding) and a procedure *Allowed Sum of Delays Propagation* (*ASDP*). Naturally, before calling the procedure ASDP, the sum of the anticipated delays (ATDs) which will certainly be made by all unassigned operations (no operation has been assigned at the pre-treatment phase) must be subtracted from AD. The three constraint-propagation subroutines are repeated until all subroutines reach a stable state at which there is no change on any unassigned operation's start-time domain. After that, *Solve* calls the procedure *Find*. If the call of *Find* returns *True*, *Solve* returns *True*. Otherwise *Solve* returns *False*.

```
$Solve(V_1, AD)$
  repeat
    $Two\_Consistency\_Check(V_1)$;
    $Edge\_Finding(V_1)$;
    $AD_2 := AD - New\_Anticipated\_Delays$;
    if $(AD_2 < 0)$
      return $False$;
    end if;
    $ASDP(AD_2, V_1)$;
  until (no more domain changes);
  if $(Find(V_1, AD) = True)$
    return $True$;
  else
    return $False$;
  end if;
end Procedure $Solve$
```

Figure 5.12: Solve: pre-treatment and restarts.

5.5.3 Semi-randomized heuristic for operation selection

RBS selects an appropriate operation at the current search node through calling the procedure *Find* which is given in Figure 5.13. *Find* starts with determining the minimal earliest possible finish time (*EFT*) for all unassigned operations. Then, all unassigned operations of which the earliest possible start times are smaller than *EFT* are collected in the set *CV*. These two steps are more or less like those in Nuijten's predictive-scheduling algorithm [**?**].

While *CV* is not empty, an operation *o* is randomly selected from *CV*. If there is any other operation in *CV* which is processed by the same machine with *o*, this (these) operation(s) including *o* is (are) collected in the set *SM*. Since every operation o' in the set *SM* would possibly be selected to be the next operation on the considered machine and be assigned a new start time at the current search node, a heuristic which includes a ripple-effect-estimation procedure is implemented in *Find* to evaluate the priority of these operations for decision repair. The operation in *SM* which has the minimal estimated ripple effect is then chosen at the current search node to replace the randomly selected *o*. After making the decision of operation selection, the procedure *Assign* is called. If *Assign* returns *True*, *Find* returns *True*. If *Assign* returns *False*, *Find* will put *o* back to the set of unassigned operations and increase the backtrack number *nrBT* by 1. If ($nrBT > maxBT$) holds, *Find* returns *False*. Otherwise, it backtracks to select another operation in *CV* unless it is empty.

```
$Find(V_2, AD)$
  $TV := V_2$;
  $EFT := Min\{eft(o) \mid o \in V_2\}$;
  $CV := \{o \mid o \in V_2 \wedge est(o) < EFT\}$;
  while ($CV \neq \emptyset$)
    Randomly select an operation $o (o \in CV)$;
    $MiniSD := AD$;
    $SM := \{o' \mid o' \in CV \wedge M(o') = M(o)\}$;
    if ($SM.Length > 1$)
      for ( $every\ o', o' \in SM$)
        Save ($D(o), \forall o \in V_2$);
        $EstimateSD := LookAhead\_ABSD(o', V_2, SM, V_2)$
        Recover ($D(o), \forall o \in V_2$);
        if ($EstimateSD < MiniSD \wedge EstimateSD! = -1$)
          $MiniSD := EstimateSD$;
          $o$ replaced by $o'(o := o')$;
        end if;
      end for;
    end if;
    $CV := CV - \{o\}$, $TV := TV - \{o\}$;
    if ($Assign(o, TV, AD) = True$)
      return $True$;
    end if;
    $TV := TV + \{o\}$;
    $nrBT{+}{+}$;
    if ($nrBT > maxBT$)
      return $False$;
    end if;
  end while;
  return $False$;
end Procedure $Find$
```

Figure 5.13: Find: Operation selection.

5.5.4 Making assignment or backtracking decision

The selected operation is assigned a start time in the procedure *Assign* (see Figure 5.14). Since only start-time delays are considered in the Repair-Based-Scheduling objective function, the inequation $Og(o) \leq est(o)$ always holds for all operations in V_1. So, the selected operation is assigned its earliest possible start time as its best start time. Any other value assignment will increase the value AD more than necessary.

Following the start-time assignment for the selected operation, *Assign* looks at the unassigned operation set. If it is empty, *Assign* returns $True$. If the unassigned operation set is not empty, *Assign* saves the start-time domains of all unassigned operations. Then, it calculates the remaining sum of the delays (AD_1) in which only the sum of the consumed delays by the already assigned operations are subtracted. Successively, *Assign* propagates the current assignment over yet unassigned operations by performing repeatedly the three constraint propagations until no domain changes anymore. If no start-time domain of an unassigned operation becomes empty after the constraint propagations, the procedure *Find* is recursively called. Otherwise, all unassigned operation start-time domains are recovered. Subsequently, the earliest possible start time of the currently selected operation is increased by $max(EFT(\{o'|o' \in V_x \wedge M(o') == M(o)\}), est(o) + 1)$ time units and then *Assign* returns $False$, i.e., RBS backtracks to select another operation.

The repeatedly performed constraint propagations in the procedure *Assign* includes *Two Consistency Check*, *Edge Finding* and *ASDP*. Before calling the subprocedure *ASDP* which propagates maximum allowed sum of delays, the new maximum allowed delays for yet unassigned operations, AD_2, is calculated. It is the result of the count in which the new ATDs of all unassigned operations are subtracted from AD_1. If AD_2 is less than or equal to zero, *Assign* returns $False$. Otherwise, AD_2 is propagated over yet unassigned operations by calling the procedure *ASDP*.

5.6 Ripple-effect estimations

In this section, we first introduce the basic ripple-effect estimation which is adopted in the procedure *Find*. Then, we present several design choices in the RBS in which the basic ripple-effect estimation and the *semi-randomized* heuristic are extended.

5.6.1 Basic ripple-effect estimation

In RBS, the ripple-effect-estimation procedure is named $LookAhead\_ABSD(o'$, $Unassigned\_set$, $Ordering\_set$, $Fixed\_unassign\_set)$. The pseudo code is given in Figure 5.15 and the procedure is called by procedure *Find*. Here, the set SM is transferred to the parameter $Ordering\_set$. The procedure is used to estimate the ripple effects of the candidate operations in the set SM for repairing the random

```
$Assign(o, V_x, AD)$
  $ST(o) := est(o)$;
  if ($V_x = \emptyset$)
    return $True$;
  else
    Save ($D(o'), \forall o' \in V_x$);
    $AD_1 := AD - Already\_Consumed\_Delays$;
    repeat
      $Two\_Consistency\_Check(V_x)$;
      $Edge\_finding(V_x)$;
      $AD_2 := AD_1 - New\_Anticipated\_Delays$;
      if ($AD_2 < 0$)
        return $False$;
      end if;
      $ASDP(AD_2, V_x)$;
    until (no more domain changes);
    if ($\forall o' \in V_x, D(o') \neq \emptyset$)
      if ($Find(V_x, AD) = True$)
        return $True$;
      end if;
    else
      recover $D(o')(\forall o' \in V_x)$;
      $est(o) := max(EFT(\{o' | o' \in V_x \wedge M(o') = M(o)\}), est(o) + 1)$;
    end if;
  end if;
  return $False$;
end Procedure $Assign$
```

Figure 5.14: Assign: Making an assignment or a backtracking decision.

selection decisions. When an operation $o'(o' \in SM)$ is temporarily selected, the idea is to estimate its ripple effect with respect to the minimal λ value of the perturbation function. Thereafter o' will be tentatively assigned its earliest possible start time as its start time and then it will be removed from $Unassigned\_set$ and $Ordering\_set$. Subsequently, the tentative assignment of o' is propagated to $Unassigned\_set$ by traditional constraint propagations (*Two Consistency Check* and *Edge Finding*). If in the current $Unassigned\_set$, there is an operation of which the domain becomes empty, the *LookAhead* procedure returns -1. Otherwise, the *LookAhead* procedure proceeds with checking the current $Ordering\_set$. When this set becomes empty, for all operations in the $Fixed\_unassign\_set$, the differences between their temporarily formed earliest possible start times and their assignments in the original schedule are summed and returned as the estimated λ value. If the $Ordering\_set$ is not empty, another operation o'' in the $Ordering\_set$ is selected and its start time is tentatively assigned to be its earliest possible start time. The tentative assignment is propagated by calling recursively the routine *LookAhead*. In order to select an o'' from $Ordering\_set$, a minimal finish time sw_1 for all operations in the $Ordering\_set$ is calculated. Then o'' is randomly selected from a subset of the current $Ordering\_set$ in which all operations' *est* are less than or equal to sw_1.

Note that the *LookAhead* procedure implicitly uses the start-time domains associated with the unassigned operations. Any change on the start-time domains resulted from the recursive calls of *LookAhead* will be recovered in the procedure $Find$.

5.6.2 Enlarging the set of tentatively assigned operations

In some machine-breakdown instances, a large percentage of operations must participate in the repair. So, at the beginning and in the early stages of the RBS search process (closer to the root of a search tree), more operations need to be considered for ripple impacts. Since the goal is far away, and since the number of the tentatively assigned operations in the basic ripple-effect estimation is quite small in proportion to all unassigned operations, the impacts of these tentative assignments on the earliest possible start times of the unassigned operations are quite small and limited. Thus, in the early stages of the RBS search processes, the information provided by the basic ripple-effect estimation is quite incomplete and imprecise. The possibility for RBS to make a wrong choice is increased. Furthermore, if a wrong selection is made in an early stage, backtracking to correct the selection becomes pretty hard in a large search tree.

Under the circumstances in which a large percentage (more than 80 percent) of operations participate in the repair, the *semi-randomized* operation selection heuristic may not work as well as we expected for the above reasons. (We will show the performance evaluation in the next chapter.) So, our attention was focus on adapting the basic ripple-effect-estimation procedure. Let us therefore reexamine the procedure $LookAhead\_ABSD(o', Unassigned\_set, Ordering\_set,\ Fix\_unassigned\_set)$

```
$LookAhead\_ABSD(o', Unassigned\_set, Ordering\_set, Fix\_unassigned\_set)$
  retval:=0;
  $FV := Unassigned\_set$;
  $FV.remove(o')$;
  $ODS := Ordering\_set$;
  $ODS.remove(o')$;
  if $(FV \neq \emptyset)$
    Forward_checking$(o', est(o'), FV)$;
    repeat
      $Two\_consistency\_check(FV)$;
      $Edge\_finding(FV)$;
    until (no domain changes);
    if $(\exists o(o \in FV), D(o) = \emptyset)$
      return -1;
    else if $(ODS = \emptyset)$
        for $(every\ o, o \in Fixed\_unassign\_set)$;
          $retval := retval + est(o) - Og(o)$;                    (1)
        end for;
    else
        select an operation $o^\star$ from $ODS$;
        $sw_1 := est(o^\star) + p(o^\star)$;
        for $(each\ o, o \in ODS)$
            if $(est(o) + p(o) < sw_1)$
              $sw_1 := est(o) + p(o)$;
            end if;
        end for;
        $CV_1 := \{o \mid o \in ODS, est(o) \leq sw_1\}$;
        randomly select $o''$ from $CV_1$;
        $retval := LookAhead\_ABSD(o'', FV, ODS, Fixed\_unassign\_set)$;
    end if;
  end if;
  return $retval$;
end Procedure $LookAhead_ABSD$
```

Figure 5.15: LookAhead: Estimating the sum of delays.

presented in Figure 5.15. It has four parameters. When $LookAhead\_ABSD$ is called by $Find$, the parameter o' will probe the operations in the set SM one by one. No alternatives for this parameter could be considered. The parameter $Unassigned\_set$ and $Fix\_unassigned\_set$ are the same set at the moment when $LookAhead\_ABSD$ is called by the procedure $Find$. The former is used to remove the tentatively assigned operations from the set and to propagate the tentative assignments to the still remaining operations. The latter is used to calculate the sum of estimated delays for all unassigned operations when the current procedure $Find$ is called. The set is not changed during the tentative assignment processes. If we removed some operations out of the two sets, the ripple-effects-estimation would be on a subset of all currently unassigned operations. Thus, the estimation is not a global but a local estimation. That estimation would not provide much help for considering the global impacts of an operation selection. So, we remark that, the alternatives of these two sets are not considered adequately for their impact on the ripple effect.

The third parameter, $Ordering\_set$, takes the set SM in $Find$. Each operation in the set is tentatively assigned a start time. We tried to transfer a large set to this parameter. Namely, enlarging the operation set in which the operations need to be tentatively assigned a start time in the ripple-effect-estimation procedure. The idea is to have a wider picture of the ripple effects by tentatively assigning more operations. For this purpose, the set CV was taken into account as the basic *semi-randomized* heuristic extension. Thus the set CV is transferred to the parameter $Ordering\_set$ of the procedures $LookAhead\_ABSD$. The RBS algorithm which uses this *semi-randomized* heuristic extension is denoted as RBS1 hereafter. The performance of this extended heuristic will be evaluated in the next chapter.

5.6.3 Extending the ripple-effect estimation

Tentatively assigning more operations may not bring more precise information for estimating ripple effects and the ripple-effect-estimation procedure itself will spend more time. However, further enlarging the $Ordering\_set$ would become too arbitrary. Therefore, we attempted another way to extend RBS1 and to improve its performance.

As indicated in Subsection 5.6.2, the earlier the machine breaks down, the more operations must participate in the repair and the more difficult it is for RBS to make a right choice in the early stage of the search process. Estimating the sum of delays of all unassigned operations (in RBS), even after enlarging the tentative assignment set (in RBS1), would not provide precise guidance for the earlier search processes. However, there is another kind of heuristic that could be exploited for guiding the search processes.

Apparently, if the new start time of an operation is equal to its original start time, its start-time delay is zero. The more operations have that their new start times are equal to their original start times, the smaller the sum of the start-time delays of these operations is. Thus, it is likely (but not certain) that the sum of the start-time delays of all unassigned operations becomes smaller. In the search process, this knowledge, which enables more unassigned operations to maintain their original start times in their start-time domains, could be exploited as a weak heuristic to select an operation. Therefore, a small probability is designed to grant this heuristic for selecting an operation as the next operation on the implicitly selected machine. To estimate this kind of ripple effects, a new procedure, called *LookAhead_NUMB*, is developed. The majority of the pseudo codes of the procedure *LookAhead_NUMB* is the same as that of the procedure *LookAhead_ABSD* (cf. Figure 5.15). Only line (1) in the middle of the procedure *LookAhead_ABSD* (cf. Figure 5.15) needs to be replaced by the line:

$$if(Og(o) == est(o)) \quad retval := retval + 1;$$

The procedure *LookAhead_NUMB* estimates and returns the number of the operations of which the earliest possible start times are equal to their original start times. The set CV is transferred to the parameter *Ordering_set* of the procedure *LookAhead_NUMB*.

To estimate the ripple effects in terms of the λ value and the NC (Equation 5.4) value, the newly extended *semi-randomized* heuristic consists of two ripple-effect-estimation procedures *LookAhead_ABSD* and *LookAhead_NUMB*. However, only a small probability is granted to estimate ripple effects through calling procedure *LookAhead_NUMB*. The majority of probabilities is granted to estimate ripple effects through calling procedure *LookAhead_ABSD*. In both procedures, the set CV is transferred to the parameter *Ordering_set*. The adapted RBS algorithm with the newly extended *semi-randomized* heuristic is denoted as RBS2. Except for the procedure *Find*, the pseudo codes of the RBS2 are the same as those of the RBS. The pseudo codes of the new *Find* procedure (denoted as *Find*2) is presented in Figure 5.16, where the *probaval* is less than 0.5.

5.7 Procedure for propagating allowed sum of delays

Before launching the search and during the search process, RBS executes three constraint-propagation procedures which includes *Two Consistency Checking*, *Edge Finding* and *Allowed Sum of Delay Propagation*(see Subsection 5.5.2 and Subsection 5.5.4). *Two Consistency Checking* and *Edge Finding* procedures are the traditional constraint-propagation techniques used in predictive scheduling. Due to reasons of efficiency, we have implemented Nuijten's *Edge Finding* algorithm for calculating LB_{est} and UB_{lct} [?]. Details on this implementation can be found in Nuijten's work [?]. In Figure 5.17, we present the *Allowed Sum of Delay Propagation* (*ASDP*) procedure.

```
$Find2(V_2, AD)$
  $TV := V_2$;
  $EFT := Min\{eft(o) \mid o \in V_2\}$;
  $CV := \{o \mid o \in V_2 \wedge est(o) < EFT\}$;
  while $(CV \neq \emptyset)$
    Randomly select an operation $o(o \in CV)$;
    $MiniSD := AD$;
    $MaxNC := 0$;
    $SM := \{o' \mid o' \in CV \wedge M(o') = M(o)\}$;
    if $(SM.Length > 1)$
      if $(random\ [0,1] > probaval)$
        for $(every\ o', o' \in SM)$
          Save $(D(o), \forall o \in V_2)$;
          $EstimateSD := LookAhead\_ABSD(o', V_2, CV, V_2)$
          Recover $(D(o), \forall o \in V_2)$;
          if $(EstimateSD < MiniSD \wedge EstimateSD! = -1)$
            $MiniSD := EstimateSD$;
            $o$ replaced by $o'(o := o')$;
          end if;
        end for;
      else
        for $(every\ o', o' \in SM)$
          Save $(D(o), \forall o \in V_2)$;
          $EstimateNC := LookAhead\_NUMB(o', V_2, CV, V_2)$
          Recover $(D(o), \forall o \in V_2)$;
          if $(EstimateNC > MaxNC)$
            $MaxNC := EstimateNC$;
            $o$ replaced by $o'(o := o')$;
          end if;
        end for;
      end if;
    end if;
    $CV := CV - \{o\}$, $TV := TV - \{o\}$;
    if $(Assign(o, TV, AD) = True)$
      return $True$;
    end if;
    $TV := TV + \{o\}$;
    $nrBT := nrBT + 1$;
    if $(nrBT > maxBT)$
      return $False$;
    end if;
  end while;
  return $False$;
end Procedure $Find2$
```

Figure 5.16: New *semi-randomized* heuristic for operation selection.

Let V_1 be the set of operations that must participate in the repair, AD be the maximum allowed sum of delays for the start times of all unassigned operations, $o_{i,j}$ be the operation in ith job at the jth position ($o_{i,j-1} \prec o_{i,j}$), *gap* be an array to represent the gap (slack) between two adjacent operations on one job. Here, the sum of the ATDs and the delays by already assigned operations have be subtracted from AD. The AD is assumed to be completely consumed by the start-time delays of the operations on one job. The $ASDP$ algorithm presented in the Figure 5.17 is based on the calculations introduced in Subsection 5.4.4. It determines the latest possible start times of all operations participating in the repair after imposing the new AD as a unary constraint.

5.8 Chapter conclusions

This chapter presented an explorative study for addressing the second part of the general research problem given in the problem statement (Section 1.2). To cope with the changes caused by an unexpected event (e.g., a machine breakdown) occurring in a job shop, a new CSP model for the repair-needed JSSP has been proposed as well as a novel Repair-Based Scheduling algorithm (RBS), which is developed under the framework of constraint-directed search. An innovative operation-selection (semi-randomized) heuristic based on ripple-effect estimation for a scheduling decision and a new constraint-propagation technique, which imposes the optimization needs on the operations' start-time domains, form the key components of the algorithm RBS. To make our Repair-Based Scheduling approach more efficient and robust, different design choices for the ripple-effect estimations of the operation-selection heuristic are proposed. The effects of the design choices will be evaluated in the next chapter.

```
$ASDP(AD, V_1)$
  for (each job $J_i$, $i = 1$ till the number of jobs in the JSSP)
    for (each $o_{i,j}$ in $J_i$, $j$ take value from the smallest to the largest)
      if ($o_{i,j}$ is the first operation in $J_i$)
        $gap[i][j] := 0$;
      else
        $gap[i][j] := est(o_{i,j}) - (est(o_{i,j-1}) + p(o_{i,j-1}))$;
      end if;
    end for;
  end for;

  for (each job $J_i$, $i = 1$ till the number of jobs in the JSSP)
    for (each $o_{i,j}(o_{i,j} \in V_1)$ in $J_i$, $j$ from the smallest to the largest)
      $sg_1 := 0, wsg_1 := 0$;
      for (each $o_{i,k}$, from $o_{i,j}$ to the last operation in $J_i$)
        if ($o_{i,k}$ is the last operation in $J_i$)
          $temp := est(o_{i,j}) + sg_1 + \lfloor (AD - wsg_1)/(k - j + 1) \rfloor + 1$;
          $lst(o_{i,j}) := min(lst(o_{i,j}), temp)$;
          break;
        else if ($wsg_1 < AD <= wsg_1 + (k - j + 1) * gap[i][k + 1]$)
          $temp := est(o_{i,j}) + sg_1 + \lfloor (AD - wsg_1)/(k - j + 1) \rfloor + 1$;
          $lst(o_{i,j}) := min(lst(o_{i,j}), temp)$;
          break;
        else
            $wsg_1 := wsg_1 + (k - j + 1) * gap[i][k + 1]$;
            $sg_1 := sg_1 + gap[i][k + 1]$;
        end if;
      end for;
    end for;
  end for;
end Procedure $ASDP$
```

Figure 5.17: Algorithm for latest possible start times.

Chapter 6

Performance Evaluation

In this chapter, we report the performance of our approaches for solving repair-needed JSSP instances. The important parameters of a repair-needed (machine-breakdown) JSSP instance are discussed and determined in Section 6.1. Section 6.2 presents the experimental results by using RBS to solve simple machine-breakdown instances. In Section 6.3, the experimental results of different design choices of the RBS, viz. RBS1 and RBS2 discussed in Subsection 5.6.2 and Subsection 5.6.3, are compared with that of RBS. Section 6.4 deals with the experiments on the important parameters of RBS2, which include the number of restarts, the backtracking factors and the probability partition factors in the extended *semi-randomized* heuristic. The experimental results of other alternative operation-selection heuristics are presented in Section 6.6 and compared with the schedules generated by RBS2. The issue about the comparison with other already developed reactive scheduling systems is discussed in Section 6.7. More machine-breakdown instances of the JSSPs solved by RBS2 are presented in the Appendix.

6.1 A repair-needed JSSP instance

As mentioned in Chapter 5, a repair-needed JSSP instance in our experiments originates from a standard JSSP in which a machine breaks down for a period of time. To carry out the experiments for solving such repair-needed JSSPs, we use a number of the instances of standard JSSPs as introduced by Fisher and Thompson [?], Carlier [?], Lawrence [?], Adams [?] and Yamada [?]. They are considered as the given JSSPs.

To deal with machine-breakdown problems as broadly as possible, a machine-breakdown instance of a given JSSP is randomly generated in our experiments. As mentioned in Chapter 5, three random integers are required to generate randomly a machine-breakdown instance. The first random integer is required to select randomly a machine as the breakdown machine in the given JSSP. It is independently and uniformly sampled from the interval $[0, NM)$ (Machines in a standard JSSP are numbered starting with 0), where NM denotes the number of machines in the given JSSP. The second random integer is required to determine the breakdown time of the selected machine. In our experiments this random integer is independently and uniformly sampled from the interval $[0, \frac{MP}{2}]$, where MP denotes the make-span of the original schedule of the JSSP. It means that, when a machine breakdown occurs, more than half a part of the original schedule has not been executed for the generated instances. If the breakdown time is greater than $\frac{MP}{2}$, the majority of the operations has been executed and thus the generated problem instance would be easier (since a smaller number of operations participate in the repair) than the ones we considered in the experiments. So, the limitation for this integer guarantees that the generated machine-breakdown instances do not represent the simple instances. The third integer is required to anticipate the duration of the machine failure, that is the time period between the machine breakdown and its recovery (or another machine which can do the same job as the broken one is in place). This integer is assumed to be equal to 20 percent of the make-span of the original schedule $(20 * \frac{MP}{100}\rfloor)$. The reason for setting such anticipating duration is to generate appropriate machine-breakdown instances from a given JSSP. If the anticipating duration is too short, such as shorter than some gaps (slacks) between the originally scheduled finish time and the start time of the two adjacent operations on the same machine, the set V_1 in Equation 5.6 would be empty. Thus the repair is not necessary for the generated instance and the work is done. If the anticipating duration is too long, generating a new schedule which has a minimal difference with the original one may lose its significance. After getting the duration of the machine failure, the recover time of the breakdown machine is obtained ($t_{recover} = t_{failure} + duration$). The number of the breakdown machine, the breakdown, and the recovery time are called the *three principal parameters* of a repair-needed (machine-breakdown) JSSP instance.

Note that the different seeds for the generator of a random number will generate different machine-breakdown instances for a given JSSP. In order to investigate machine-breakdown instances to a large extent and evaluate the performances of RBS, RBS1 and RBS2, we adopted different seeds in the experiments to generate a number of different repair-needed (machine-breakdown) JSSP instances for a given JSSP.

6.2 Solving simple machine-breakdown instances

In order to evaluate conveniently the performances of RBS and its alternatives RBS1 and RBS2, the number of restarts, the backtracking factor and the probability partition factor are fixed (in this section and in the next section). A maximum number of 500 restarts is allowed for all iterations; $\lfloor 0.2 * N \rfloor$ backtracking steps (0.2 is the

6	6				
0	1	**16**	**22**	**42**	**49**
0	8	**13**	**28**	**38**	**48**
1	***6***	**10**	**18**	**27**	**30**
8	**13**	**22**	**29**	**37**	**45**
13	**22**	**25**	**38**	**48**	**52**
13	**16**	**19**	**28**	**45**	**49**

Figure 6.1: An original schedule of FT06.

backtracking factor and N is the number of the operations in the given JSSP) is used in each search tree; probability 0.1 and 0.9 are granted to call the procedures *LookAhead_NUMB* and *LookAhead_ABSD* respectively. The influences of the different number of restarts, backtracking factors and probability partition factors for RBS and its alternative designs will be dealt with in Section 6.4.

Except for an explicit indication, all tests in this thesis are performed on a Pentium-450MHz PC.

Let us first investigate a machine-breakdown instance originated from FT06 (6 jobs and 6 machines; FT means that the JSSP instance was introduced by Fisher and Thompson [?]). An original schedule of FT06 which has the optimal make-span 55 is presented in Figure 6.1. Line 1 contains the number of jobs and the number of machines; followed by 6 lines (for each job one line) listing the operations' start times of the original schedule in an ascending order, respectively. A machine-breakdown instance of FT06 denoted as $FT06_a$ has the following *three principal parameters*: machine 3 breaks down at time 2 and recovers at time 13. According to the analysis described in Section 5.3, the *Key*-operation is identified as the second operation of the third job which is denoted by the bold italic number in the Figure 6.1. There are 30 operations participating in the repair. The bold numbers in the Figure 6.1 denote the corresponding start times of the operations that participate in the repair. Henceforth, the same notations will be used for showing the original schedules of other JSSP instances.

Figure 6.2 shows a new schedule obtained by the RBS algorithm. In this figure, OG denotes that the corresponding operation does not need to participate in the repair. The number followed by a symbol "=" means that the new start time assignment of the corresponding operation made by RBS is equal to its original start time. Henceforth, the same notations will be used for showing a repaired schedule of another machine-breakdown instance. The λ value of the new schedule is 113, i.e., the sum of the start-time delays of the new schedule with respect to the original start times of the original schedule (Figure 6.1).

Table 6.1 shows the corresponding make-span (MP) and the λ value of three scheduling methods, where RSH denotes the right shift scheduling; SMP denotes the scheduling algorithm developed by Nuijten [?] for approximating optimal make-span. In comparison with the schedules generated by RSH and SMP, the schedule

OG	OG	16=	22=	44	54
OG	OG	13=	34	44	54
OG	13	17	25	34	35
OG	13=	22=	29=	42	51
13=	22=	25=	47	54	58
13=	17	25	34	50	54

Figure 6.2: A new schedule of $FT06_a$ with the minimal λ value.

Schedule / Objective	RSH	SMP	RBS
MP	62	60	60
λ	210	192	113

Table 6.1: Different schedules of $FT06_a$.

generated by RBS has a much smaller λ value.

Since FT06 is an easy JSSP instance which consists of a small number of operations, the corresponding machine-breakdown instances are easy too. One may argue that the techniques which are used for solving these instances may not be suitable for the machine-breakdown instances of a JSSP which consists of a large number of operations. Therefore, we turn to investigate a machine-breakdown instance of FT10 (10 jobs and 10 machines) which is a very difficult JSSP instance for obtaining a predictive schedule with an optimal make-span.

An original schedule of FT10 with a make-span 985, which is obtained by Nuijten's algorithm [**?**] (the edge finding is implemented by calculating LB_{est} and UB_{lct}) with a higher frequency, is presented in Figure 6.3. A machine-breakdown instance of FT10 denoted as $FT10_a$ has the following *three principal parameters*: machine 8 breaks down at time 308 and recovers at time 505. The *Key*-operation is identified as the eighth operation of the seventh job. There are 68 operations participating in the repair.

Figure 6.4 shows a new schedule obtained by using RBS. The λ value of the perturbation function for the new schedule is 3366, i.e., the sum of the start-time delays of the new schedule with respect to the original start times of the original schedule (Figure 6.3) is 3366 time units.

Table 6.2 shows the corresponding make-span (MP) and the λ values of three scheduling methods. In comparison with the schedules produced by RSH and SMP, the schedule produced by RBS has a much smaller λ value.

Although RBS is an approximate algorithm which uses a randomized restart

10	10								
205	**472**	**559**	**596**	**650**	**699**	**710**	**812**	**868**	**952**
234	**469**	**575**	**650**	**661**	**730**	**772**	**818**	**868**	**940**
381	**472**	**557**	**596**	**670**	**760**	**775**	**818**	**907**	**952**
147	228	323	**407**	**516**	**596**	**654**	**782**	**880**	**902**
0	113	274	375	**436**	**506**	**575**	**605**	**715**	**907**
45	145	147	250	**345**	**393**	**557**	**613**	**780**	**787**
0	76	113	174	187	219	240	***393***	**482**	**725**
14	119	228	301	375	**525**	**648**	**667**	**739**	**880**
0	76	174	250	323	**465**	**476**	**516**	**699**	**760**
296	**394**	**408**	**469**	**482**	**546**	**622**	**730**	**786**	**940**

Figure 6.3: An original schedule of FT10

OG	472=	559=	596=	650=	699=	710=	829	888	952=
OG	469=	575=	650=	661=	730=	772=	818=	943	1015
OG	472=	557=	596=	709	799	817	883	973	1018
OG	OG	OG	407=	516=	638	690	782=	880=	902=
OG	OG	OG	OG	436=	506=	617	638	781	972
OG	OG	OG	OG	569	661	733	818	883	918
OG	OG	OG	OG	OG	OG	OG	799	888	918
OG	OG	OG	OG	OG	525=	690	733	781	880=
OG	OG	OG	OG	OG	465=	476=	516=	699=	932
OG	394=	408=	469=	505	569	645	730=	786=	1015

Figure 6.4: A schedule of $FT10_a$ with λ=3366.

Schedule / Objective	RSH	SMP	RBS
MP	1145	1044	1060
λ	10880	5013	3366

Table 6.2: Different schedules of $FT10_a$.

strategy in the framework of iterative improvement, it generates quite stable solutions for the machine-breakdown instances $FT06_a$ and $FT10_a$. Actually, in our experiments, every run of RBS algorithm generates a new schedule with the same λ value for $FT06_a$ and $FT10_a$. Furthermore, RBS only takes two or three restarts to get a schedule for the two machine-breakdown instances. So, RBS is fast and effective to solve $FT06_a$ and $FT10_a$. Is RBS still fast and effective for solving other machine-breakdown instances of the JSSPs?

Let us examine a machine-breakdown instance $LA09_a$ which is originated from a JSSP instance LA09 (LA means that the JSSP instance was introduced by Lawrence [?]). The $LA09_a$ has the following *three principal parameters*: machine 3 breaks down at time 166 and recovers at time 356. The *key*-operation is identified as the second operation of the third job. There are 55 operations participating in the repair. The initial λ value (produced by RSH) is 10450. However, the instance $LA09_a$ is somewhat different from the instances $FT06_a$ and $FT10_a$. In this instance, the execution of the *key*-operation is interrupted by the breakdown machine. So, a unary constraint that specifies the uncompleted part of the *key*-operation must start at time 356 and should be added to the constraint set of $LA09_a$. Moreover, its process time should be adjusted to the duration of the uncompleted part of the *key*-operation (according to Equation 5.8).

15	5			
17	**279**	**364**	**448**	**567**
0	83	179	**685**	**832**
0	***126***	**329**	**575**	**652**
84	**509**	**689**	**755**	**829**
0	128	**225**	**643**	**737**
135	**727**	**815**	**861**	**887**
116	162	**238**	**576**	**857**
0	147	**434**	**529**	**567**
67	192	**352**	**587**	**829**
266	**311**	**570**	**652**	**772**
396	**529**	**628**	**706**	**861**
0	198	**364**	**586**	**746**
88	**206**	**279**	**402**	**500**
59	**500**	**568**	**693**	**706**
98	**244**	**329**	**445**	**602**

After running 10 times of RBS algorithm to solve $LA09_a$, the biggest and smallest λ values among all generated schedules are 5344 and 5257 respectively. The average λ value of 10 repaired schedules is 5331.9. This means that RBS does not always generate the schedules with the same λ value when it is used to solve more complex machine-breakdown JSSP instances. The similar phenomena can be observed from the algorithms which adopted a randomized strategy to generate a predictive schedule for an optimal (near-optimal) make-span. For example, Vaessens, Aarts and Lenstra [?] compare 20 of the best approaches to the JSSP for 13 instances. Ranking the approaches that generate a predictive schedule according to effectiveness, Nuijten's [?] approach (using a randomized strategy) comes in the sixth place.

The deviation of the lowest make-span found by Nuijten's approach from the best known lowest make-span is on average 2.26%.

In Subsection 5.6.2 and Subsection 5.6.3, we introduced two design alternatives of RBS, RBS1 and RBS2. Both of them are designed to extend the basic *semi-randomized* heuristic of RBS for improving its performance in terms of stability (smaller deviations between the results of different executions) and optimality (smaller λ value).

In the next section, we will present some experimental results to compare the performance of RBS with its alternative designs.

6.3 Performances of RBS, RBS1, and RBS2

Let us start with examining a more complex machine-breakdown instance $FT10_b$ which also originated from JSSP instance FT10. The original schedule of FT10 with a make-span 985 is the same as the one presented in Figure 6.3. The *three principal parameters* of the $FT10_b$ are as follows: machine 3 breaks down at time 118 and recovers at time 315. The *key*-operation is identified as the third operation of the seventh job. There are 88 operations participating in the repair. In this instance, the execution of the *key*-operation is also interrupted by the breakdown machine. The initial λ value (the sum of delays produced by RSH) is 17336. By running 10 times of RBS algorithm to solve $FT10_b$, each run finds a solution. The average λ value of the 10 repaired schedules is 7092.2.

In the instance $FT10_b$, a large percentage (88 percent) of the operations must participate in the repair while 68 percent and 73 percent of the operations must participate in the repair in the instances $FT10_a$ and $LA09_a$, respectively. So, at the beginning and in the early stages of the RBS search process (closer to the root of a search tree), more operations in the $FT10_b$ than in the $FT10_a$ need to be considered for ripple impacts. To guide the search making a adequate decision in the early stages of a search process, two alternative designs of the RBS, which are introduced in Subsection 5.6.2 (RBS1) and Subsection 5.6.3 (RBS2), are developed for solving such complex machine-breakdown instances.

By running 10 times the RBS1 and RBS2 respectively to solve the machine-breakdown instances $FT06_a$, $LA09_a$, $FT10_a$ and $FT10_b$, respectively, the average λ value of the generated schedules are presented in Table 6.3. Compared with the schedules obtained by RBS, we can conclude that RBS1 performed better than RBS for the instances $LA09_a$ and $FT10_b$, and RBS1 performed as well as RBS for the instances $FT06_a$ and $FT10_a$. However, slightly more restarts were needed for RBS1 than RBS. RBS2 performed slightly better than RBS1 for the instances $LA09_a$, and as well as RBS1 (and RBS) for the instances $FT06_a$ and $FT10_a$, and considerably better than RBS1 (and RBS) for the instance $FT10_b$.

For simple repair-needed JSSP instances, RBS2 performed roughly the same as RBS and RBS1. However, for complex repair-needed JSSP instances, RBS2 per-

Instances / Algorithms	LA09a	FT06a	FT10a	FT10b
RBS	5331.9	113	3366	7092.2
RBS1	5273.6	113	3366	7077.2
RBS2	5189.1	113	3366	6005.1

Table 6.3: Comparison of extended *semi-randomized* heuristics.

formed much better than RBS and RBS1 for some instances and slightly better than RBS1 for other instances. By synthesizing their performances, we chose RBS2 as the representative of our Repair-Based Scheduling algorithms in our further studies.

OG	396	481	565	627
OG	OG	OG	685=	880
OG	356	396	575=	713
OG	509=	689=	766	895
OG	OG	225=	935	1019
OG	738	835	914	980
OG	OG	238=	576=	905
OG	OG	840	942	980
OG	OG	352=	587=	1019
266=	311=	595	652=	820
396=	554	628=	754	881
OG	OG	627	698	797
OG	481	627	737	846
OG	500=	568=	700	797
OG	244=	329=	445=	697

6.4 Important parameters in RBS2

In this section, we investigate the influence of three main parameters of RBS2, namely (1) the backtracking factor BTF, (2) the number of restart *nrRestarts* (cf. Figure 5.11) in the *iterative_improve* procedure, and (3) the probability partition factor *probaval* (cf. Figure 5.16) in the extended *semi-randomized* operation selection heuristic. For each parameter setting, we performed ten runs of RBS2.

6.4.1 Backtracking factors

In the following experiments we vary the value for the backtracking factor BTF of Subsection 5.5.1. We chose to use $BTF = 0$, 0.2, 0.4, 0.6, 0.8, 1.0, 2.0, 4.0. Observe

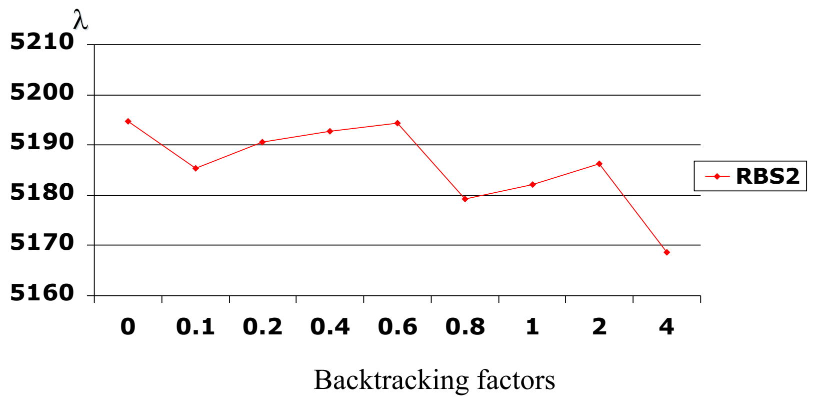


Figure 6.5: Average λ value of RBS2 for $LA09_a$, $nrRestarts$=500.

that $BTF = 0$ implies that each time a dead end was found, a restart is performed. When we changed the BTF value, the number of restarts remained fixed to 500 and the probabilities to call the procedures $LookAhead\_NUMB$ and $LookAhead\_ABSD$ for ripple-effect estimations were fixed to 0.1 and 0.9, respectively.

The experimental results by running RBS2 which is coupled to different backtracking factors when solving $LA09_a$, are shown in Figures 6.5 and 6.6. Figure 6.5 shows the corresponding average λ values and Figure 6.6 shows the corresponding average CPU times.

It can be seen that with the increasing of BTF, the performance of RBS2 does not improve uniformly. However, the time cost is steadily increased with the BTF value increasing. For $BTF = 0$, i.e., no chronological backtracking is used at all, the performance is clearly worse. From these experiments we conclude that any value for BTF between 0.10 and 0.30 is reasonable for balancing the time cost of RBS2.

6.4.2 Number of restarts

A reasonable number of restarts is important for finding a schedule with a good λ value in a certain time period. In the following experiments we vary the value for the number of restarts $nrRestarts$ of Subsection 5.5.1. We chose to use $nrRestarts$ = 200, 400, 600, 800, 1000, 1200, 1400. When we changed the $nrRestarts$ value, the backtracking factor BTF was fixed to 0.2 and the probabilities to call the procedures $LookAhead\_NUMB$ and $LookAhead\_ABSD$ were fixed to 0.1 and 0.9, respectively.

The experimental results by running RBS2 which is coupled to a different number of restarts when solving $LA09_a$, are shown in Figures 6.7 and 6.8. Figure 6.7 shows the corresponding average λ values and Figure 6.8 shows the corresponding average CPU times.

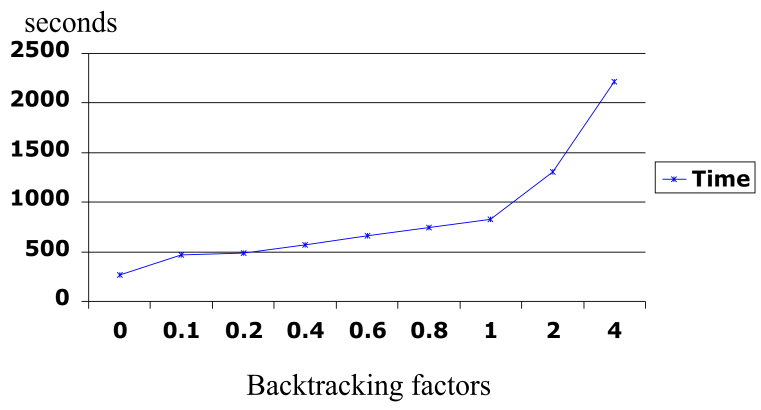


Figure 6.6: Average time of RBS2 for $LA09_a$, $nrRestarts$=500.

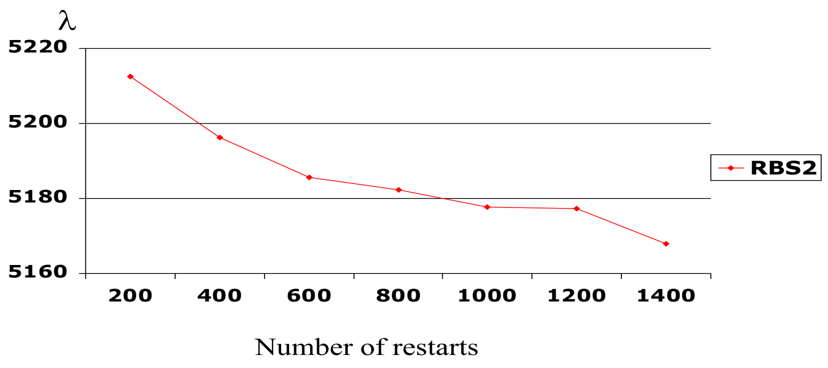


Figure 6.7: Average λ value of RBS2 for $LA09_a$, $BTfac$=0.2.

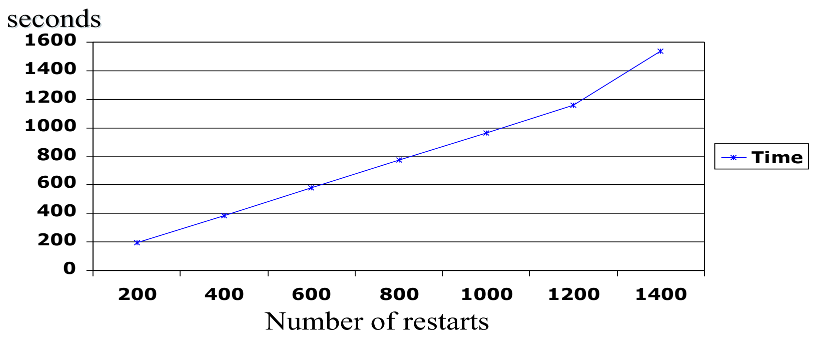


Figure 6.8: Average time of RBS2 for $LA09_a$, $BTfac$=0.2.

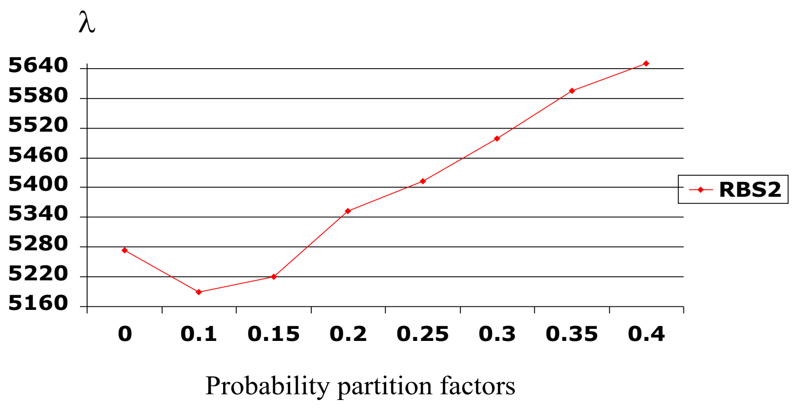


Figure 6.9: Average λ value of RBS2 for $LA09_a$.

It can be seen that with the increasing number of restarts, the performance of RBS2 improves steadily, i.e., the average λ value of the schedules decreases steadily. However, the time cost is linearly increasing with the increasing number of restarts. From these experiments we conclude that a moderate number of restarts is required for balancing the time cost of RBS2. For real repair-needed JSSPs, the number of restarts can be adjusted according to the constraint which specifies the time at which the repaired schedule must be ready.

6.4.3 Probability partitions

In the following experiments we varied the value for the probability partition factors $probaval$ of the procedure $Find2$ in Subsection 5.6.3. We chose to use $probaval =$ 0, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40. Observe that $probaval = 0$ implies that the procedure $LookAhead\_NUMB$ is not called, and the $LookAhead\ \_ABSD$ is called with probability 1 for ripple-effect estimation. When we changed the $probaval$ value, the number of restarts $nrRestarts$ was fixed to 500 and the backtracking factor BTF was fixed to 0.2.

The experimental results obtained by running RBS2, which is coupled to different probability partition factors for calling ripple-effect-estimation procedures when solving $LA09_a$, are shown in Figures 6.9 and 6.10. Figure 6.9 shows the corresponding average λ values and Figure 6.10 shows the corresponding average CPU times.

It can be seen that RBS2 gave the best performance with $probaval = 0.1$ and a gradually decreasing performance for $probaval$ values greater than 0.15. For $probaval = 0$, the performance is clearly worse. This means that calling the procedure $LookAhead\_NUMB$ with zero (without calling) or a larger (greater than 0.15) probability would hinder RBS2 to find a better schedule. From these experiments

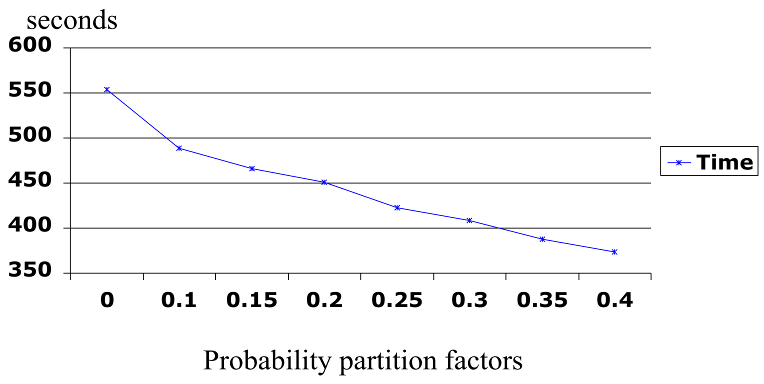


Figure 6.10: Average time of RBS2 for $LA09_a$.

and the experiments presented in Section 6.3, we conclude that a small probability can be used to call the procedure $LookAhead\_NUMB$ for balancing the performance of RBS2 when solving either moderate or very difficult repair-needed JSSPs.

6.5 Additional experimental results

To test the performance of our Repair-Based Scheduling approach (RBS2) on more repair-needed JSSP instances, we chose nine JSSP instances: FT06, FT10, LA09, LA30, CAR1, CAR5, ABZ5, ORB9 and YAM1. These instances are the representatives of standard JSSPs as introduced by Fisher and Thompson [?], Carlier [?], Lawrence [?], Adams [?] and Yamada [?]. For each chosen JSSP instance, five different repair-needed JSSP instances were generated. We used the algorithm RBS2 to solve these instances in which the number of restarts ($nrRestarts$) and the backtracking factor (BTF) are set to 500 and 0.2, respectively. The probability partition factor $probaval$ of the procedure $Find2$ is 0.1. For each repair-needed JSSP instance, we performed ten runs using different seeds for the random number generator.

The results of the tests for the repair-needed JSSPs originated from FT10 are given in Table 6.4. The column "JSSP" of Table 6.4 gives the JSSP instance name and the number of jobs and machines. The column "MP" gives the make-span of an original schedule. The column "Instance" indicates different repair-needed JSSP instances originated from FT10. The column "bdm" gives the number of the breakdown machine. The columns "t_f" and "t_r" give the machine breaking and recovering time, respectively. The column "n" gives the number of operations that must participate in the repair. The column "ini_SD" gives the initial sum of delays (λ value generated by RSH). The columns "avg_SD" and "T" give the average sum of delays (λ value) and the average running time (seconds) over ten runs, respectively.

JSSP	MP	Instance	bdm	t_f	t_r	n	ini_SD	avg_SD	T	Δ
FT10	985	FT10a	6	308	505	68	10880	3366	249	0
(10*10)		FT10b	3	118	315	88	17336	6005.1	319	1.05
		FT10c	0	343	540	66	13002	6224.4	160	0.12
		FT10d	6	147	344	84	13188	1296	192	0
		FT10e	5	341	538	71	13987	3713.2	154	0.16

Table 6.4: Performance of RBS2 for FT10.

The column "Δ" gives the deviation of the maximum found sum of delays from the minimum found sum of delays in terms of percentage of the minimum found sum of delays.

It can be seen that the deviation of the maximum found sum of delays from the minimum found sum of delays is on average 0.266%; the running time is on average 214.8 seconds. This leads to the conclusion that RBS2 performs well in terms of efficiency and solution stability.

For other repair-needed JSSPs, the results obtained by RBS2 are given in the **Appendix A** and the same notations as above will be used for showing the experimental results for those repair-needed JSSP instances.

6.6 Alternative design choices

In the empirical studies, we explored a large number of different designs to construct a variety of search trees (spaces) for Repair-Based Scheduling. These approaches include the operation-selection alternatives in RBS (keeping other parts of RBS unchanged) and the completely different strategies to construct a search tree (space) (such as selecting a machine at first and then ordering all operations processed by the machine, Simulated Annealing etc.).

For the approaches in the first category, three alternatives are investigated: (1) randomly selecting an operation from the set CV without using the ripple-effect estimation; (2) selecting an operation which has the lowest ripple-effect estimation in the set CV; and (3) first randomly selecting a machine from the machines on which the operations in the set CV are processed, and then selecting an operation processed by the selected machine which has the minimum estimated ripple effects. In testing these alternatives, the number of restarts ($nrRestarts$) and the backtracking factor (BTF) are set to 500 and 0.2 respectively.

For the first alternative (denoted as Alter1), the main drawback is that the search may wander off, away from the objective in the search processes. By running the modified algorithm ten times, the average λ value of the generated schedules are presented in Table 6.5. We can see that, for $LA09_a$, $FT10_a$, and $FT10_b$, the results

Instances / Algorithms	LA09a	FT06a	FT10a	FT10b
Alter1	5444.1	113	3500.4	6553.5
Alter2	5274.3	113	3366	6209.1
Alter3	5188.5	113	3366	6024.2
RBS2	5189.1	113	3366	6005.1

Table 6.5: Average sum of delays of alternatives.

are much worse than that of RBS2. Apparently, it cannot be used to obtain a stable and near-optimal solution for complex machine-breakdown instances. These observations inspired us to develop a new heuristic to select an operation in the search processes.

For the second alternative (denoted as Alter2), the main drawback is that every operation in the set CV is evaluated with respect to the ripple effects at a search node, and that the time to make an operation-selection decision increases considerably. So, much more time is required to generate a new schedule by this approach than by RBS2. This may violate the practical constraints that require the reactive scheduling to generate a new schedule as soon as possible. Moreover, the search processes in the algorithm would become relatively more deterministic than in RBS2. Restarting the search a number of times would search through a smaller part of search space than RBS2 does. As a result, a better solution would not be found for some problems. In the modified algorithm Alter2, the probability partition factor granted to estimate ripple effects through calling the procedure $LookAhead\_NUMB$ and the procedure $LookAhead\_ABSD$ is the same as in RBS2. After running ten times Alter2 to solve the machine-breakdown instances, the average λ value (sum of the delays) of the generated schedules are presented in Table 6.5 in comparison with the average λ value of RBS2. It can be seen that the schedules generated by Alter2 are not too bad with respect to the schedules generated by RBS2 in terms of λ value. However, the average running time of Alter2 in solving $FT06_a$, $LA09_a$ and $FT10_a$ is three or four times as RBS2 needs, in solving $FT10_b$ it is 5 to 6 times as RBS2 needs. So, Alter2 has no practical significance.

For the third alternative (denoted as Alter3), the machines which process the operations in the current set CV have the same probability of being selected. However, in contrast with RBS2, the more operations a machine process in the current CV, the higher probability a machine has to be selected. The probability partition factor granted to estimate the ripple-effect of an operation through calling the procedure $LookAhead\_NUMB$ and the procedure $LookAhead\_ABSD$ is the same as in RBS2. By running the modified algorithm ten times, the average λ value (sum of the delays) of the generated schedules is presented in Table 6.5. Although the time cost of Alter3 is roughly the same as that of RBS2, its solution result in terms of the λ value is worse than RBS2 in solving $FT10_b$ and roughly the same in solving $LA09_a$. So, it is not considered to be a better operation-selection heuristic.

Some of the approaches in the second category which use completely different strategies to construct a search tree (space) (such as selecting a machine at first and then ordering all operations processed by the machine) are also constraint-based strategies. However, the search is resource (machine) oriented. It means that an appropriate machine is selected at first and then all operations processed by the selected machine are ordered and assigned sequentially. We have tried the heuristic which selects a machine that has the largest number of start-time conflicts and then order operations on the selected machine. But the algorithm fails to generate a schedule for the instances $FT10_a$ and $LA09_a$ within the same time period in which RBS2 runs 500 restarts. Then we tried the heuristic which is developed by Baptiste [**?**] to generate a schedule with the minimal make-span. It works as follows. For the operations participated in the repair, it first determines that their earliest and latest start and finish times are globally consistent with all the temporal constraints. Then, a machine among the machines required by unordered operations is selected and all operations that require the selected machine are sequenced to satisfy the resource (machine) constraints. The algorithm can generate a schedule for $FT10a$ and $LA09a$, but the result is much worse than that of RBS2 in terms of the λ value. In combination with the above two machine-selection heuristics, we attempted a large number of other heuristics to sequence the operations on the selected machine. However, none of them significantly improved the performances of the approaches.

6.7 Comparison with other methods

In Section 3.7, we mentioned that many reactive-scheduling systems were developed in the past decade. Although the reactive-scheduling systems currently in existence claim to emphasize on keeping the minimal changes to the original schedule, most of them primarily try to balance this objective with the traditional optimization objectives, such as minimizing make-span, work-in-process (WIP) inventory, mean tardiness of jobs etc [**?**]. However, our Repair-Based Scheduling approach emphasizes the importance of keeping the continuity of execution and the real-time response to reduce the disruption in the original schedule. The original schedule is modified to the minimum extent possible by reaching a specific objective function (Equation 5.5). Since an iterative improvement over the λ value is adopted, a new schedule can be obtained in the time window in which the schedule must be ready.

As pointed out in [**?**], comparing the performance of the different reactive-scheduling methods (algorithms) is far from straightforward, because there are many dimensions along which an algorithm's performance can be measured. For instance, in a particular application, the most important performance measures are probably run time and solution optimality. Unfortunately, the run time of program could be affected by many factors, including the choice of programming language, the data structure, the programming style, the machine used, etc. Some methods (algorithms) could be implemented more efficiently in one language than in another, and comparing the run time of programs written in different languages is not very significant in general.

Since different objectives are pursued and different constraint relaxations are allowed in reactive-scheduling systems, comparing RBS (RBS2) with other reactive-scheduling systems is not meaningful at present. For instance, in OPIS, the performance was evaluated with respect to weighted criteria reflecting optimization, stability, and efficiency objectives, where the optimization goal is to balance weighted tardiness and WIP minimization. The goals of CABINS are (1) to arrive at a schedule that does not violate any constraint, (2) to optimize the modified schedule according to the user's preference, and (3) to minimize the schedule disruption. However, only the outcome of experiments which reflect the user's preference for minimizing weighted tardiness and minimizing the combination of weighted tardiness and WIP are presented. Yet the "Probe Backtrack Search" developed by Sakkout [**?**, **?**] claims to reconfigure minimally the schedules in response to a changing environment. Nevertheless, the approach was developed to handle specific application problems only. In the benchmark problems they solve, the duration of an activity is a variable and the resources available for activities are allowed to be reduced in the dynamic changing environment. The general aim of their approach includes to re-assign activity start times to reduce the number of resources needed by the schedule. That aim is not pursued in our Repair-Based Scheduling approaches.

Some research focused on generating *robust schedules* (*solutions*) [**?**, **?**] that are likely to remain valid after minor changes to the problem. Nevertheless, major changes (for instance machines breakdown) usually make the modifications inevitable to very robust schedules.

Since the difficulty in comparison, no experimental evidence has been provided so far in favor of incremental schedule repair as opposed to rescheduling. Like for predictive scheduling problems, no polynomial-time solution algorithm is known for reactive-scheduling problems. The computation times for reactive scheduling are usually highly variable and unpredictable [**?**]. In order to bridge the gap between capabilities and requirements, a reactive-scheduling system is considered as real-time response only in the simplest sense in which the program runs fast enough to cope with the job shop events. As shown in the experimental results, the Repair-Based Scheduling approach developed in our research well balances the response time and the solution optimality in terms of the sum of delays with the originally scheduled start times.

6.8 Chapter conclusions

This chapter presented the experimental results of our Repair-Based Scheduling approaches. We conclude that the basic Repair-Based Scheduling algorithm (RBS) works quite well for solving simple repair-needed JSSP instances in terms of solution quality (with respect to the λ value) and the computational time cost. However, RBS did not always generate the schedules with the same λ value when it was used to solve more complex machine-breakdown JSSP instances. For this reason, two design alternatives of RBS, RBS1 and RBS2 were examined. We found that RBS2 had roughly the same performance as RBS and RBS1 in solving simple repair-needed JSSP instances. For complex repair-needed JSSP instances, RBS2 performed much better than RBS and RBS1 for some instances and slightly better than RBS1 for other instances. After comparing the performances, we chose RBS2 as the representative of our Repair-Based Scheduling algorithms.

Then, we investigated the influence of three main parameters of RBS2, namely (1) the backtracking factor BTF, (2) the number of restart *nrRestarts* (cf. Figure 5.11) in the *iterative_improve* procedure, and (3) the probability partition factor *probaval* (cf. Figure 5.16) in the extended *semi-randomized* operation selection heuristic. We arrived at three conclusions: (1) any value for BTF between 0.10 and 0.30 is reasonable for balancing the time cost of RBS2; (2) a moderate number of restarts is required for balancing the time cost of RBS2 (for real repair-needed JSSPs, the number of restarts can be adjusted according to the constraint which specifies the time at which the repaired schedule must be ready); and (3) a small probability can be used to call the procedure $LookAhead\_NUMB$ for balancing the performance of RBS2 when solving either moderate or very difficult repair-needed JSSPs. Scrutinizing additional experimental results, we concluded that RBS2 performed well in terms of efficiency and solution stability.

Finally, in this chapter we showed some experimental results of three alternative design choices for Repair-Based Scheduling. These alternative designs either cannot find good solutions or take too much time to obtain a similar solution in terms of λ value as RBS2 arrived at. So, they are not considered as good as Repair-Based Scheduling approaches as RBS2.

In closing we would like to remark that comparing RBS (RBS2) with other reactive-scheduling systems is not meaningful at present, since completely different objectives are pursued and different constraint relaxations are applied in reactive-scheduling systems currently in use.

Chapter 7

Concluding Remarks

In this thesis we addressed the general research problem given in the problem statement (Section 1.2), and developed a range of repair-based approaches for solving DCSPs and repair-needed JSSPs; both may arise from a dynamically changing environment. The DCSPs are the result of constraint additions in a series of CSPs, and the repair-needed JSSPs are caused by unexpected events (machine breakdown) occurring in a job shop. The problems of solving DCSPs and repair-needed JSSPs are NP-hard. Our investigations led to new solution methods for these problems in the field of AI. The principal issues characterized in the problem statements (Section 1.2) were successfully addressed and a number of innovative repair-based algorithms under the framework of constraint-directed search were developed accordingly. The experimental results showed that the resulting algorithms are capable of dealing with the computational complexity in a reasonable way and generate high quality solutions with respect to the specific objective functions.

We remark that our main conclusions are already given in Chapters 4 and 6. In this final chapter we confine ourselves to concluding remarks, a list of contributions, and some future research directions. We start making some concluding remarks on repair-based approaches for DCSPs (Section 7.1) and on repair-based scheduling approaches (Section 7.2). We list our contributions in Section 7.3 and provide future research directions in Section 7.4.

7.1 Repair-based approaches for DCSPs

For solving DCSPs, we have proposed a complete algorithm, RB-AC, and two approximation algorithms, BS and RS. All algorithms aim at finding a solution for a CSP requiring a minimal or a near-minimal number of assignment changes with respect to its infringed solution. Although RB-AC theoretically will find a solution with a minimal number of assignment changes for a CSP (if such a solution exists) in a DCSP, we have shown that the time complexity of RB-AC is too high to solve large CSPs which consist of a large number of variables and have large domains for these variables. Therefore, two approximation algorithms were proposed to bring the computational time and the solution quality in a more balanced position. The experimental results showed that in a limited period of time both approximation algorithms can obtain solutions that are close to a solution with the minimal number of assignment changes. Moreover we showed that RS clearly outperformed BS.

Concerning the parameter combinations in RS, we concluded that independent of whether a general CSP has looser or tighter constraints, using a sufficiently large number of restarts, a moderate number of backtracks, an adequate mini-conflict value selection and an appropriate search-depth adaptation in RS is regarded as the best parameter combination for getting a solution with a near minimal number of assignment changes in solving a CSP.

Comparing RB-AC (or RS) with other repair-based methods is not straightforward. First, there exists currently no method using the same objective function as we adopted for RB-AC (or RS). Second, most of the methods are limited to a specific application domain (e.g., the methods developed by Verfaillie and Schiex [**?**, **?**]) or based on some particular assumptions which are not well-founded for changes caused by unexpected events (e.g., proposals by Wallace and Freuder [**?**]).

The exploration of Repair-Based methods in solving DCSPs is oriented towards general CSPs, in which no domain knowledge is available for guiding the search processes. However, in practical situations the domain knowledge can be quite useful for identifying the characteristics of the constraints, selecting adequately a variable or a value at a search node, and designing specific constraint-propagation techniques to prune the search space as we did in Chapter 5. Hence, when solving the real-world problem of the minimal number of assignment changes for a DCSP, the RB-AC (or RS) should be combined with specific domain knowledge. This will further improve the solution quality and reduce the computational time cost.

7.2 Repair-based scheduling approaches

In exploring Repair-Based Scheduling methods for repair-needed JSSPs caused by machine-breakdown events in a job shop, the key characteristics of the repair-needed JSSPs were first identified and analyzed. Then, the essential model modifications (constraint additions and relaxations) on the original JSSP model and the optimal objective to be achieved by Repair-Based Scheduling activities were broadly investigated and clearly designated. Concerning the model modifications, we demonstrated which operations can be safely excluded from repair activities; and which constraints must be added to the set of constraints of the original JSSP model under different circumstances. Subsequently, a new CSP model of a repair-needed JSSP and an innovative approximate algorithm (RBS2) based on the framework of constraint-directed search were designed and built.

In Subsection 5.2.4, the objective of Repair-Based Scheduling was set to generate a new schedule that has a minimal sum of start-time delays with the original schedule (this was defined by a perturbation function, Equation 5.5). Since the objective is completely different from those of predictive scheduling, new approaches with a variety of architectures, methodologies and tools were extensively investigated and explored. The development of our Repair-Based Scheduling approach has been influenced by previous work on predictive job shop scheduling. This influence can be observed from three structural aspects in our approach: (1) the construction of the search tree (under the framework of constraint-directed search); (2) the adoption of a randomized restart; and (3) the iterative improvement of the requirement specified by the objective function. Our approach deliberately combines the architectural design of a search algorithm with seeking intelligently a specific solution that meets the demands of Repair-Based Scheduling. In detail, we developed a sophisticated operation-selection (semi-randomized) heuristic and a novel constraint-propagation technique, which imposes the optimization needs for Repair-Based Scheduling, being the key components of the approximation algorithm RBS2. As the experimental results shown in Chapter 6 and the Appendix, the approximation algorithm RBS2 is able to obtain a good quality solution with a low computational time cost.

In the search process of RBS2, first the candidate operations which will likely be selected at a search node are determined. Then, the semi-randomized heuristic is invoked; the heuristic is largely dependent on the estimation of the ripple-effect (with respect to the sum of the start-time delays) of the candidate operations. Since directly estimating the sum of the start-time delays in an early stage of a search process does not work well for a large repair-needed JSSP, an indirect estimation which estimates the maintenance of the original start-time assignments, is carried out with a small probability included in the heuristic. Considering both the efficiency of the search processes and the approximation of the optimal solution, only a limited number of the candidate operations are evaluated with ripple-effect at a search node and only a part of constraint-propagation techniques (two consistency and edge-finding) are carried out in the estimation procedure. By using the semi-randomized heuristic, the search process in RBS2 is thus opportunistically directed. Therefore, different executions of RBS2 need not to produce the same solutions. However, the deviations with respect to the best solution found, are small. As pointed out by Jain and Meeran [**?**], approximation methods do not guarantee achieving exact solutions. They are, however, able to obtain near-optimal solutions, within moderate computing times and are therefore more suitable for larger problems. The importance of approximation methods is indicated by Glover and Greenberg [**?**] who suggest that direct tree searching is rather unsatisfactory for combinatorially difficult problems. They indicate that heuristics inspired by natural phenomena and intelligent problem solving are most suitable in bridging the gap between operations research and artificial intelligence.

The new constraint-propagation technique (propagating the allowed sum of delays) was developed through analyzing and capturing the dependencies and correlations between the optimization needs (achieving the minimal sum of the start-time delays) and the possible start-time delay of an individual operation. The precedence constraints between operations of a job and the assumption that the operations on one job are responsible for all the delays, form the basis of the technique. In this constraint-propagation technique, the allowed sum of delays for all unassigned operations is imposed on these operations' start-time domains as a unary constraint. The allowed sum of delays excludes not only the real consumed sum of delays caused by already assigned operations, but also the anticipated sum of delays which consists of unassigned operations' delays in future search processes.

The parameter tuning of RBS2 for balancing the solution quality and time cost in solving either moderate or very difficult repair-needed JSSPs were investigated in Chapter 6. There, we concluded that (1) a small backtrack factor is adequate; (2) a moderate number of restarts is required, and (3) a small probability can be granted for indirect ripple-effect estimation. For solving real repair-needed JSSPs, it is better to fix the backtrack factor and the probability partition factor, and to change (increase or decrease) the number of restarts according to the time window in which the repaired schedule must be ready.

The next section (Section 7.3) summarizes the main contributions of this thesis and the Section 7.4 suggests future research directions based on the limitations of the current methods.

7.3 Contributions

The main contribution of this thesis is the development of two sets of new repair-based methodologies and techniques in the field of AI, viz. for adequately solving DCSPs and repair-needed JSSPs. By implementing the corresponding designed algorithms, we demonstrated that they are capable of repairing effectively successive CSPs of a DCSP and repair-needed JSSPs with a reasonable computation-time cost.

The second contribution is establishing a CSP model for a repair-needed JSSP when an unexpected event, i.e., a machine-breakdown, occurs in a job shop. The thesis presents an extensive and detailed analysis that facilitates the problem modeling and problem-solving configurations.

The third contribution is the appropriate exploitation of the quantified objective of Repair-Based Scheduling, i.e., guiding the search process and pruning the operations' start-time domains. This is done by the development and application of a novel semi-randomized operation-selection heuristic which is based on the direct and indirect ripple-effect estimation and a new constraint-propagation technique.

The Repair-Based Scheduling approach developed in our research is different from other reactive-scheduling systems mainly in the following four points: (1) our approach reschedules all operations which participate in the repair; (2) the repair activities are carried out on operations according to the precedence constraints, viz.

the repair starts with (and persists in) the operations of which the earliest possible start times are in front of the earliest possible finish times of the repair-needed operations; (3) the repair decisions are made by considering the global (and not the local) ripple-effect of the repair activities; and (4) optimization needs are utilized to form a heuristic guiding the search processes and imposing a unary constraint on the unassigned operations' start-time domains.

The algorithms and solution methodologies developed in this research have successfully demonstrated that it is possible to develop efficient procedures for Repair-Based Scheduling in the field of AI.

7.4 Future research

There are a number of possible extensions to the work described in this thesis. As pointed out in Sections 5.1 and 5.2, the Repair-Based Scheduling is one of reactive-scheduling approaches which generates a new schedule that is minimally different from the original schedule. It thus can be integrated into an automated scheduling system which is capable of responding to a variety of events in a dynamically changing environment and which will meet multi-criteria objectives.

Since dynamically changing environments invariably present different challenges, there is a considerable diversity of problem characteristics along several dimensions for reactive scheduling. We mention: (1) in the structure of different domains, (2) in the types of constraints added, (3) in the performance objectives and preferences that must be attended to, and (4) in the types of uncertainties that must be accommodated. The problem characteristics along each of these dimensions might dominate the model construction, the scheduling heuristic and the design of solution procedure. Therefore, an automated scheduling system should address these issues so that it can adequately deal with various types of uncertainties that exist in scheduling problems and consider multiple criteria which describe various performance measures of schedules. For instance, additional reactive-scheduling approaches are needed to respond to the unexpected events, such as arrival of rush orders, which require that the new start times of some operations are earlier than their existing start times.

Such extensions of the scheduling work need the attention of three research themes (scheduling or reactive scheduling, fuzzy reasoning, multi-criteria decision making) with the specific aim of successfully investigating difficult, uncertain and dynamic real-world scheduling problems. Additional research is required in the area of problem-solving method configuration. For instance, the integration of incremental repair methods with our Repair-Based Scheduling methods is another avenue of research worth pursuing.

Experimental results

The results of the tests for the repair-needed JSSPs originating from FT06, LA09, LA30, CAR1, CAR5, ABZ5, ORB9 and YAM1 are given in Table 1.

- The column "JSSP" of Table 1 gives the JSSP instance name and the number of jobs and machines.
- The column "MP" gives the make-span of an original schedule.
- The column "Instance" indicates 5 different repair-needed JSSP instances originating from each JSSP instance.
- The column "bdm" gives the number of the breakdown machine.
- The columns "t_f" and "t_r" give the machine breaking and recovering time, respectively.
- The column "n" gives the number of operations that must participate in the repair.
- The column "ini_SD" gives the initial sum of delays (λ value generated by RSH).
- The columns "avg_SD" and "T" give the average sum of delays (λ value) and the average running time (seconds) over ten runs, respectively.
- The column "Δ" gives the deviation of the maximum sum of delays found from the minimum sum of delays found in terms of percentage of the minimum sum of delays found.

JSSP	MP	Instance	bdm	t_f	t_r	n	ini_SD	avg_SD	T	Δ
FT06	55	FT06a	2	12	23	29	319	154	24	0
(6*6)		FT06b	1	16	27	21	231	71	5	0
		FT06c	0	10	21	23	184	88	6	0
		FT06d	3	2	13	30	210	113	42	0
		FT06e	5	15	26	23	253	150	17	0
LA09	951	LA09a	3	166	356	55	10450	5189.1	265	0.94
(15*5)		LA09b	1	310	500	47	8930	3160	120	0
		LA09c	4	154	344	57	10830	3041	207	0.65
		LA09d	4	202	392	53	10070	2636	168	0.76
		LA09e	0	290	480	45	8550	2110	76	0
LA30	1374	LA30a	1	239	513	160	43840	8994	1894	3.69
(20*10)		LA30b	0	634	908	98	26852	5312.9	447	0.16
		LA30c	6	258	532	155	42470	8660.1	1459	7.08
		LA30d	3	171	445	175	47950	15112.2	3869	7.41
		LA30e	5	641	915	88	24112	5631.3	327	0.83
CAR1	7038	CAR1a	3	2599	4006	29	40455	11797	31	0
(11*5)		CAR1b	0	1370	2777	42	59094	40727.7	84	1.49
		CAR1c	1	3361	4768	19	22078	15861	13	0
		CAR1d	1	789	2196	43	60501	29866.9	84	2.39
		CAR1e	0	1185	2592	42	59094	40840.1	83	2.05
CAR5	7767	CAR5a	3	2074	3627	43	46655	21864	64	0
(10*6)		CAR5b	4	2304	3857	41	48872	6035	39	0
		CAR5c	1	1149	2702	53	82309	61855.2	158	2.58
		CAR5d	0	2032	3585	49	76097	40235	132	0
		CAR5e	1	1549	3102	46	71208	45849	120	0
ABZ5	1242	ABZ5a	3	529	777	44	10076	3158	39	0
(10*10)		ABZ5b	0	253	501	76	18848	4793	181	0
		ABZ5c	6	173	421	79	19592	9412.1	307	2.03
		ABZ5d	1	500	748	57	14316	5460.6	82	0.54
		ABZ5e	5	275	523	70	16450	7800.2	224	1.8
ORB9	937	ORB9a	3	178	365	75	12975	4893.8	190	2.47
(10*10)		ORB9b	0	235	422	73	13651	6778	166	0
		ORB9c	6	229	416	59	7139	2195	88	0
		ORB9d	1	116	303	76	11324	3018.8	206	1.13
		ORB9e	5	401	588	50	9350	2388	64	0
YAM1	988	YAM1a	3	490	687	177	34869	9734.9	2157	1.84
(20*20)		YAM1b	10	339	536	245	48265	9303	2094	3.11
		YAM1c	6	177	374	302	59192	11478.9	10499	3.31
		YAM1d	1	205	402	282	55554	11300.1	4989	3.68
		YAM1e	15	271	468	256	46336	8539.4	5056	5.84

Table 1: Performance of RBS2 for more repair-needed JSSPs .

Summary

This thesis presents research in the field of AI (Artificial Intelligence) on repair-based approaches for DCSPs (Dynamic Constraint Satisfaction Problems) and repair-needed JSSPs (Job Shop Scheduling Problems). In Chapter 1 we provide some background information and formulate the problem statement. We start with the introduction of notions, definitions, and representations of (1) a DCSP which is derived from a sequence of CSPs, and (2) a repair-needed JSSP which is the result of an unexpected event (e.g., a machine breakdown). The goal in dealing with a DCSP is to find a minimal number of assignment-change solutions for successive CSPs. The goal in solving a repair-needed JSSP is to find a new schedule with a minimal sum of start-time delays. Hence, the problem statement is twofold and reads: (1) Is it possible to develop new methods that adequately solve DCSPs? and (2) Is it possible to develop new methods that adequately solve repair-needed JSSPs?

In Chapter 2, the main subject is adequately handling DCSPs. Some important issues on solving a CSP are discussed. These issues include the formal definition of a CSP, complexity of a CSP, search methods for solving CSPs, constraint-propagation techniques used to prune the search space, variable-selection and value-selection heuristics for guiding the search, dead-end handling techniques, and optimization in CSPs and Dynamic CSPs. Since CSPs are NP-hard problems, the derived DCSPs are NP-hard problems too.

In Chapter 3, a formal definition of a JSSP and a useful CSP model of a JSSP are presented. Then, a number of important notations in job-shop scheduling are provided. Moreover, some powerful methods which have successfully been used for solving JSSPs in the AI community are discussed and investigated. These methods include constraint-based scheduling approaches, local-search algorithms and genetic algorithms. Particular attention will be paid to the constraint-based scheduling approaches which use a constraint-directed search framework for solving JSSPs. We mentioned that the success of the constraint-based scheduling mainly can be attributed to the following factors: (1) utilizing the constraint representations to model the problem knowledge; (2) guiding the search process by operation selection heuristics and start-time selection heuristics; (3) pruning the search space by using constraint-propagation techniques; and (4) combining the techniques developed in

Operation Research (OR) (such as edge-finding [?] etc.). Subsequently, the algorithms based on constraint-directed search for constructing a job-shop schedule with an optimal or a near-optimal make-span are discussed. These algorithms include an optimization algorithm (Baptiste [?]), approximation algorithms (Applegate and Cook [?], Nuijten [?]) and an ORR-FSS algorithm (Sadeh and Fox [?, ?]). Moreover, a set of constraint-propagation techniques adopted from the job-shop scheduling domain are introduced, which include a time-table propagator, a disjunctive constraint propagator, as well as edge-finding and energy-based reasoning techniques. Finally, the concepts of predictive and reactive scheduling are examined. The rescheduling and incremental repair strategies for reactive scheduling are briefly discussed and a number of typical reactive-scheduling systems are mentioned.

In Chapter 4, three repair-based methods for solving DCSPs are proposed. They are a complete repair-based algorithm (RB-AC) and two approximation algorithms (BS and RS) (RB-AC stands for Repair Based - Arc Consistency, BS for Binary Search, and RS for Restart). First, the necessity of finding a solution with a minimal number of assignment changes for a CSP is given. Then, the idea and methodology behind the method are presented. Subsequently, a complete repair-based algorithm (RB-AC) which combines local search and constraint-propagation techniques is proposed and the termination, correctness, completeness, and optimality of RB-AC are proved. Following the analysis of the time complexity of RB-AC, two approximation algorithms are developed to obtain a solution with a near-minimal number of assignment changes. After a series of empirical studies, we conclude that the approximation algorithm RS outperforms BS. Finally, we compose a best parameter combination for RS.

Chapter 5 presents a broadly explorative study for Repair-Based Scheduling in the field of AI. First, the motivation of conducting a repair for an existing schedule is clarified. Then, we discuss the following four issues: (1) why an innovative Repair-Based Scheduling approach is needed? (2) what kind of model modification should be made to the original JSSP model? (3) what objective should be achieved in Repair-Based Scheduling? and (4) what minimal perturbation function should be used? Thereafter, we make a thorough analysis for an unexpected event (e.g., a machine breakdown) that may occur in a job shop. We build a new CSP model for the repair-needed JSSP, by identifying which operations need to participate in repair and what constraints should be added to the original constraint set. Subsequently, we outline a novel Repair-Based Scheduling algorithm (RBS) which is developed under the framework of constraint-directed search for handling the machine-breakdown event. The main ideas, related techniques and the pseudo codes of the RBS are presented and illustrated in the chapter. Finally, the procedures which form the main part of an innovative heuristic and the different design choices for RBS are presented. A new constraint-propagation technique is developed and an algorithm to implement this technique is shown. The new technique imposes the optimization needs on the operations' start-time domains and thus helps to prune the search space.

Chapter 6 reports on the performance of our Repair-Based Scheduling approaches in solving repair-needed JSSP instances. We start to explain how to generate a repair-needed JSSP instance (caused by a machine breakdown) from a standard JSSP instance fitting our experiments. The important parameters of a machine-breakdown JSSP instance are discussed and determined. Subsequently, we present experimental results by using RBS to solve simple machine-breakdown instances. Thereafter, three algorithms being the results of different design choices: RBS, RBS1 and RBS2, are evaluated on simple, moderate, and difficult machine-breakdown instances. The experimental results obtained with these algorithms for different problems are compared. As a result, RBS2 is selected as the best representative of our Repair-Based Scheduling approaches. The important parameters in RBS2 which include the number of restarts, the backtracking factors, and the probability partition factors in the extended semi-randomized heuristic are evaluated for different values in a variety of experiments. The impact of these parameters to RBS2 concerning efficiency and optimality are clearly exhibited by the experimental results. Results of other alternative operation-selection heuristics are presented too, and compared with the schedules generated by RBS2. Finally, the issue about the comparison with other existing reactive-scheduling systems is discussed.

Chapter 7 contains some concluding remarks on the methodologies followed. Most importantly, it summarizes our three contributions: (1) developing two sets of new repair-based methodologies and techniques for DCSPs and JSSPs, (2) establishing a new CSP model for repair-needed JSSPs, and (3) taking full advantage of the quantified objective of repair-based scheduling for the development of a novel semi-randomized operation-selection heuristic and a new constraint propagation technique. The contributions are an adequate answer to the problem statement in Section 1.2. However, not all problems are solved by now. Chapter 7 finishes with showing some future research directions, such as reactive scheduling in combination with fuzzy reasoning and multi-criteria decision making.

Samenvatting

Dit proefschrift beschrijft een vierjarig onderzoek op het gebied van AI (Artificial Intelligence) over het onderwerp: reparatiegebaseerde benaderingen van DCPSs (Dynamische CSPs; CSP staat voor Constraint Satisfaction Problem) en reparatiebehoevende JSSPs (Job Shop Scheduling Problems). In hoofdstuk 1 wordt de achtergrond van het probleem behandeld en het onderzoeksdoel geformuleerd. Allereerst worden begrippen, definities en representaties van een DCSP en een reparatiebehoevende JSSP geïntroduceerd. Het doel bij een DCSP is het vinden van oplossingen voor de achtereenvolgende CSPs die de DCSP vormen, waarbij de achtereenvolgende oplossingen minimaal van elkaar verschillen. Het doel van het oplossen van de reparatiebehoevende JSSP is het vinden van een nieuw rooster waarin de som van de vertragingen van de individuele operaties minimaal is. Derhalve is het onderzoeksdoel tweeledig: (1) is het mogelijk om nieuwe adequate methoden voor het oplossen van DCSPs te ontwikkelen? en (2) is het mogelijk om nieuwe adequate methoden voor het oplossen van reparatiebehoevende JSSPs te ontwikkelen?

Hoofdstuk 2 richt zich op DCSPs. Belangrijke begrippen bij het oplossen van een DCSP worden besproken. Deze begrippen betreffen de formele definitie van een CSP, de complexiteit van een CSP, zoekmethoden voor het oplossen van een CSP, constraint-propagatie-technieken voor het verkleinen van de zoekruimte, heuristieken voor de keuze van een variabele en waarde waarmee het zoekproces gestuurd kan worden, het adequaat hanteren van doodlopende zoekpaden, en optimale oplossingen van een CSP en een DCSP. Daar een CSP een NP-moeilijk probleem is, geldt hetzelfde voor een DCSP.

Hoofdstuk 3 begint met een formele definitie van een JSSP en een bruikbaar CSP-model van een JSSP. Verder wordt een aantal belangrijke begrippen van een roosterprobleem behandeld. Ook wordt een aantal krachtige technieken die succesvol zijn toegepast door de AI-gemeenschap bij het oplossen van een JSSP nader onderzocht en besproken. Deze technieken zijn onder andere constraint-gebaseerde methoden voor het roosteren, lokale zoekmethoden, en genetische algoritmen. In het bijzonder wordt aandacht besteed aan constraint-gebaseerde methoden die constraint-gerichte zoekmethoden toepassen voor het oplossen van een JSSP. Het succes van deze methoden is voornamelijk te danken aan de volgende vier factoren: (1) probleemkennis modelleren met behulp van constraints, (2) het sturen van het zoekproces met behulp van heuristieken voor de keuze van een variabele en een waarde, (3) het verkleinen van de zoekruimte door middel van constraint-propagatie-technieken, en (4) het combineren van deze technieken met technieken zoals die ontwikkeld zijn in het gebied van de Operations Research. De algoritmen die gebruik maken van constraint-gebaseerd zoeken en die ontwikkeld zijn voor het creëren van roosters met een optimale of bijna optimale verwerkingstijd worden besproken. Voorts wordt aandacht besteed aan het wezen van constraint-propagatie-technieken. Tot slot worden concepten van voorspellende en reactieve rooster-creatie onderzocht. Herroostering en incrementele reparatie-strategieën worden kort besproken en een aantal typische reactieve roostersystemen wordt genoemd.

Hoofdstuk 4 beschrijft drie nieuwe methoden voor het oplossen van DCSPs. Deze methoden zijn een volledig reparatiegebaseerd algoritme RB-AC (een afkorting van Repair-Based Arc Consistency), en twee benaderingsalgoritmen BS (Binary Search) en RS (Restart Search). Allereerst wordt echter de noodzaak voor het vinden van een oplossing met een minimaal aantal veranderingen ten opzichte van een vorige oplossing besproken. Vervolgens worden de ideeën achter de voorgestelde methode gepresenteerd. Hierna wordt het volledige reparatiegebaseerde algoritme RB-AC geïntroduceerd. Dit algoritme combineert lokaal zoeken met constraint-propagatie. De eindigheid, correctheid, volledigheid en optimaliteit van het algoritme wordt bewezen. Na een analyse van de tijdscomplexiteit van RB-AC worden twee benaderingsalgoritmen, BS en RS, voorgesteld. Aan de hand van experimentele resultaten wordt vastgesteld dat de prestaties van RS beter zijn dan die van BS. Met behulp van aanvullende experimenten wordt een optimale parameter-instelling voor het algoritme RS vastgesteld.

Hoofdstuk 5 beschrijft een verkennende studie van reparatiegebaseerde roostering binnen de AI. Eerst wordt een motivatie voor roosterreparatie besproken. Vervolgens wordt ingegaan op vier belangrijke vragen: (1) waarom is een reparatiegebaseerde aanpassing van een rooster noodzakelijk? (2) welke aanpassingen zijn nodig in het model van de originele JSSP? (3) welke doelen moeten worden bereikt met reparatiegebaseerd roosteren? en (4) welke doelfunctie moet worden geoptimaliseerd? Vervolgens wordt een analyse uitgevoerd van onverwachte gebeurtenissen die kunnen optreden in de uitvoering van een rooster. Een nieuw CSP model voor een reparatie-behoevende JSSP wordt opgesteld door het bepalen van de operaties die van belang zijn voor het reparatieproces en door te bepalen welke aanvullende randvoorwaarden aan de nieuwe situatie opgelegd moeten worden. Hierna wordt een nieuw reparatiegebaseerd roosteralgoritme RBS (Repair-Based Scheduling) beschreven. Twee innovatieve heuristieken voor de keuze van operaties en waarden worden verder geanalyseerd samen met een constraint-propagatie-techniek waarvan RBS gebruik maakt.

Hoofdstuk 6 rapporteert over de prestaties van het reparatiegebaseerd roosteralgoritme RBS. Eerst wordt uitgelegd hoe reparatiebehoevende testinstanties van een JSSP worden gegenereerd. Belangrijke parameters van deze instanties worden besproken. De resultaten van het oplossen van simpele instanties wordt vervolgens uitvoerig behandeld. Hierna worden drie algoritmen RBS, RBS1 en RBS2, die het resultaat zijn van drie verschillende ontwerpkeuzen, geëvalueerd aan de hand van simpele, gemiddelde en moeilijke reparatiebehoevende probleemgevallen. De hier verkregen resultaten worden onderling vergeleken, waarna RBS2 als meest geschikte algoritme wordt gekozen. Verschillende parameter-instellingen van RBS2 worden in een groot aantal experimenten geëvalueerd. De invloed van de parameter-instellingen op de efficiëntie en de optimaliteit wordt getoond aan de hand van experimenten. Vervolgens worden de resultaten die verkregen zijn door middel van alternatieve operatieselectieheuristieken, vergeleken met de roosters die met behulp van RBS2 gegenereerd zijn. Tot slot wordt ingegaan op de vergelijking met andere reactieve roosterbenaderingen die in de literatuur beschreven zijn.

Hoofdstuk 7 bevat een aantal afsluitende opmerkingen over de gevolgde methode. Het hoofdstuk geeft een samenvatting van drie bijdragen: (1) de ontwikkeling van twee verzamelingen van methoden en technieken voor DCPSs en JSSPs, (2) het opstellen van een nieuw CSP-model van reparatiebehoevende JSSPs, en (3) het volledig uitbuiten van het kwantitatieve doel van reparatiegebaseerd roosteren voor het ontwikkelen van een vernieuwende semi-willekeurige operatieselectie heuristiek en een nieuwe constraint-propagatie-techniek. De bijdragen geven een adequaat antwoord op de probleembeschrijving geformuleerd in paragraaf 1.2. Niet alle problemen zijn echter opgelost. Dit hoofdstuk besluit met het tonen van een aantal mogelijke toekomstige onderzoeksrichtingen, zoals een combinatie met 'warrig redeneren' en multi-criteria beslissingen.

Curriculum Vitae

Yong-Ping Ran was born on February 9, 1962 in Hubei Province of China. From 1978 to 1981, he studied at the Department of Mathematics, Three Gorges University, China. From 1981 to 1988, he worked as a mathematics teacher at Gezhouba branch of Hubei TV University. From 1988 to 1996, he worked as a computer programmer and database administrator in the computer centre of Gezhouba hydro-electric construction company in China.

In October 1996, he began his study at the Department of Informatics, Vrije Universiteit Brussels, Belgium. In September 1998, he successfully defended his M.Sc. thesis and received his Master degree in applied computer science.

From December 1998 to February 2003, he was employed by NWO (Netherlands Organization for Scientific Research) and worked as a research assistant (OIO) towards a Ph.D. degree at the Department of Computer Science (Institute for Knowledge and Agent Technology – IKAT), Universiteit Maastricht, the Netherlands. The research presented in this thesis was performed during this period.

SIKS Dissertatiereeks

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM) Memory versus Search in Games

1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets

1999-3 Don Beal (UM) The Nature of Minimax Search

1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems

1999-7 David Spelt (UT) Verification support for object database design

1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief

2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication

2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA) Learning as problem solving

2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A

Matter of Style

2001-6 Martijn van Welie (VU) Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics

2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems

2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT) Development and specification of virtual environments

2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 Yong-Ping Ran (UM) Repair-Based Scheduling