Solution Approaches for Solving Stochastic Job Shop and Blocking Job Shop Problems

C.H.M. van Blokland

Master Of Science Graduation Thesis

September 27, 2012

ICA-3235777

Universiteit Utrecht

Department of Computing Science,
Utrecht University,
Utrecht, The Netherlands

Supervisors:

dr. ir. J.M. van den Akker
Utrecht University

dr. J.A. Hoogeveen
Utrecht University

Preface

This thesis is the result of a ten months during research study on Job Shop Scheduling. During my two-year master education, I have learned in great detail the most important basics on scheduling, timetabling, simulation and optimization algorithms. In the lectures taught by Marjan van den Akker and Han Hoogeveen, they succeeded to set a high-level education. Also they were able to transfer their enthusiasm on scheduling and optimization.

In the beginning of my master phase, I was able to conduct a research study on the Blocking Job Shop Scheduling Problem in the form of an Experimentation Project. In this project, done together with a friend Dick Kampman, Han and Marjan agreed to supervise our progress. During the three months of researching, we held many meetings, discussing our progress and results. Next to the fact that the meetings were always pleasant and informal, our meetings were most of the time concluded with new insights and openings for further research. The final product resulted in many ideas and pointers for further investigation.

From the Experimentation Project, the basis for a masters thesis was formed. Since our mutual interest and incentive on solving Job Shop Scheduling problems, it was obvious that Han and Marjan would become the supervisors for my thesis.

During my thesis I had the ability to support Marjan in the practical elements of the course Simulation, which I enjoyed a lot. I also had a part-time job on the development of a new central system for course quality information. Although the combination of researching with a part-time job is somewhat overwhelming, our clear appointments and short term communication made sure to keep everything on track. It became clear that, even after studying several extensions to the Standard Job Shop Scheduling, there are many more interesting areas to explore. Therefore I think there is much room for colleague students to continue researching the Stochastic and Blocking Job Shop Problems.

Kevin van Blokland,
September 2012

Abstract

The Stochastic and Blocking Job Shop Scheduling problems are two extensions of the Standard Job Scheduling Problem. In the stochastic extension, processing times retrieve a probability distribution. This results in an objective function that is also a stochastic variable. In order to reliably calculate the objective value, a simulation has to be executed. When solving such problems, new complexities come into play and different solution approaches need to be developed. This thesis presents several new solution approaches that can optimize the Stochastic Job Shop Scheduling Problem. The blocking extension removes the unlimited intermediate machine buffers. When the buffers are not available, the already NP-Hard job shop becomes even harder to solve. In this thesis, additional solution approaches, built upon existing approaches, are discussed that can optimize the Blocking Job Shop Problem better. Using a randomized neighbourhood in combination with a MIP, new areas of solutions are being explored that lead to better solutions. Also a cycle analysis is given that describes the problem infeasibilities in greater detail.

Contents

1 Introduction to Job Shop Scheduling **1**

1.1 General Introduction . . . 1

1.2 Formal Definition and Notation . . . 1

1.2.1 Notation . . . 2

1.2.2 Mathematical Formulation . . . 3

1.3 Local Search . . . 5

1.3.1 Criticalpath block swap . . . 7

1.3.2 Waiting left shift . . . 8

2 Stochastic Job Shop Scheduling **9**

2.1 Introduction . . . 9

2.2 Formal definition of the Stochastic Job Shop Problem . . . 10

2.2.1 Objective Function . . . 11

2.2.2 Simulation model . . . 11

2.3 Solution Approach . . . 14

2.3.1 Classical Methods . . . 14

2.3.2 Sophisticated Methods . . . 17

2.4 Experimental setup . . . 20

2.4.1 Generation of benchmark instances . . . 21

2.4.2 Benchmarking architecture . . . 21

2.4.3 Generated Instances . . . 24

2.5 Results and Analysis . . . 26

2.5.1 Determination of Methods . . . 26

2.5.2 Quantitative Results of the Selected Methods . . . 28

2.5.3 Results of the Experiments on Scaled Instances . . . 29

2.5.4 Analysis . . . 30

2.5.5 Relative Performance of the Classical Cool Down Approach . . . 32

2.6 Conclusion and Further Research . . . 32

2.6.1 Estimation of Percentiles . . . 32

2.6.2 Lucky Ticket Theory . . . 33

2.6.3 Further Research . . . 35

3 Blocking Job Shop Scheduling **48**

3.1 Introduction . . . 48

3.2 Formal Definition of the Blocking Job Shop . . . 49

3.2.1 MIP Formulation . . . 51

3.2.2 Alternative Graph . . . 52
3.3 Cycle Analysis . . . 53
3.3.1 Dependency Graph . . . 54
3.3.2 Advanced Disjunctive Graph . . . 59
3.3.3 Equivalence of cycles in the Dependency Graph and the Advanced Disjunctive Graph . . . 62
3.4 Solution Approaches . . . 64
3.4.1 Mascis and Pacciarelli [16] . . . 64
3.4.2 Gröflin and Klinkert [8] . . . 65
3.4.3 Van den Broek [26] . . . 65
3.4.4 Simulated Schedule Observer . . . 66
3.4.5 Random Inserts . . . 72
3.4.6 Extended Random Inserts . . . 73
3.4.7 Local Search With Emergency Buffers . . . 74
3.4.8 Local Search With Fixed Swaps . . . 75
3.5 Results and Analysis . . . 76
3.6 Conclusion and Further Research . . . 77

Chapter 1

Introduction to Job Shop Scheduling

1.1 General Introduction

In many practical environments we can find scheduling problems. Nowadays, almost every person will somehow come in contact with one form of scheduling. Public transportation, product development, sport competitions, everywhere we can find some problem that relates to a scheduling or timetabling problem. Often a mathematical formulation can be found that specifically models the problem. The Job Shop Scheduling Problem in one of the formulations that represents a broad range of practical scheduling problems. Solving the Job Shop Scheduling Problem to optimality can prove to be very hard, due to the combinatorial structure. Even small instances become to hard to solve by humans. Therefore in the past decades, many approaches have been developed to optimize this problem.

The Job Shop Scheduling Problem can best be described by a factory floor, consisting of several machines, which all perform a specific task. The shop has a predetermined set of jobs to complete. Each job consists of a series of operations that need to be processed in order on a specific machine. Since the order of operations in a job is fixed, the only thing left to be determined is the processing order for individual machines. When a machine is finished processing an operation of a job, the job is placed in the unlimited buffer of the successor machine.

If the machine schedules are fixed, such that for each job a specific route through the machines has been found, several performance measures can be calculated. The most common performance measure or objective function is the makespan, determined by the time it takes to complete the last operation. In the literature the job shop, with multiple machines and a makespan objective function, is indicated by the following notation [6]:

$$J \,||\, C_{max} \tag{1.1}$$

1.2 Formal Definition and Notation

The Job Shop problem is a scheduling problem consisting of a set of jobs and a set of machines. Each job consists of a list of operations, that require processing on a pre-specified machine. In order to complete a job, all the operations that are defined for this job need to be executed and finished. For each Job Shop Scheduling problem several objective functions can be defined. The goal of a completed Job Shop Schedule is to minimize the objective function.

Furthermore there is an additional constraint stating that the operations of a job need to be processed in a given order. That is, if job j consists of operations {*Injection Molding*, *Painting*, *Drying*}, the *Injection Molding* first needs to be completely finished before the *Painting* operation can be started.

For each operation we know on what machine this operation needs to be executed. The Standard Job Shop Scheduling problem assumes that each job visits all the machines exactly once. This results in jobs where the amount of operations is equal to the amount of machines in the shop.

Since the order of operations in a job is fixed, the only thing that remains to be decided is the order of the executions on each machine. We can thus focus on fixing the execution order on each machine. A solution of the Job Shop Scheduling problem is therefore for each machine a set of operations and their explicit order, such that the chosen objective function is minimized.

1.2.1 Notation

First we will start with a formal description of the Job Shop. This notation that is introduced will be used throughout the thesis. For a Job Shop with M machines and N jobs we have the following set of variables:

$O_{j,i}$ indicates operation number i of job j, where $i = 0 \dots M-1$ and $j = 0 \dots N-1$.

$p_{j,i}$ indicates the processing time of operation $O_{j,i}$.

$machine(o_{j,i})$ indicates the index of the machine that has to process operation $O_{j,i}$ where $machine(O_{j,i}) \in 0 \dots M-1$.

$s(O_{j,i})$ indicates the start time of operation $O_{j,i}$.

$e(O_{j,i})$ indicates the finishing time of operation $O_{j,i}$.

$w(O_{j,i})$ indicates the waiting time of machine $machine(O_{j,i})$ caused by waiting before the start of operation $O_{j,i}$.

From this definition it follows that each job precisely has M operations. Furthermore we assume that for each job every machine will be visited exactly once. Also the end time of operation $O_{j,i}$, $e(O_{j,i})$, can be defined in terms of the start time and the (realization of the) processing time: $e(O_{j,i}) = s(O_{j,i}) + p_{j,i}$. Furthermore, the waiting time of operation $O_{j,i}$ is caused by the fact that its previous operation $O_{j,i-1}$ is not yet finished. If the waiting time equals zero, the machine immediately continues processing; the previous operation has in this case already finished.

For some applications to real-life optimization problems, the above definition does not suffice and therefore extensions need to be applied. A common extension is the Blocking Job Shop, further explained in Chapter 3. For instance in the Standard Job Shop when a machine has finished processing an operation, it can immediately continue to process the next operation defined by his schedule. This assumption implies the availability of unlimited buffer space or temporary space allocation. If we remove the buffer space, a machine will be blocked until the operation currently residing on the machine can be removed, to be processed by its next machine.

1.2.2 Mathematical Formulation

In order to determine a mathematical formulation of the Job Shop Scheduling Problem, we first need to identify the different objective functions that are common for this problem. The following list [20] gives an overview of these objective functions:

- Makespan (Maximum Completion Time)
- Total Weighted Completion Time
- Maximum Lateness
- Number of Tardy jobs
- Total Tardiness
- Total Weighted Tardiness

When solving the Stochastic Job Shop Problem and the Blocking Job Shop Problem we focus only on the optimization of the makespan objective function. This objective is relatively easy to determine and is widely accepted in both research and practical environments. Furthermore, the makespan objective function can be modeled by a graph formulation and a MIP.

The makespan is defined as the total length of the schedule when all jobs are finished. By determining the start times of the operations, and therefore the end times of the operations, we can calculate the operations with the largest end time. This operation is finished last and therefore determines the length of the schedule.

Disjunctive Graph Formulation

One approach for calculating the makespan objective value is by applying a graph algorithm on the disjunctive graph. The disjunctive graph is formulated in 1964 by Roy and Sussman [22]. In this graph the precedence order of the operations is fixed. The makespan is determined by calculating the longest path to the final operation in the schedule.

First the conjunctions are constructed. The conjunctions model the job constraints to make sure that an operation is not started before its predecessor operation is finished. For each operation in the Job Shop a vertex is created. Next a source vertex s and a sink vertex t are added. Between two consecutive operations $o_{j,i}$ and $o_{j,i+1}$ of job j we draw an arc $(o_{j,i}, o_{j,i+1})$. The arc is weighted by the processing time of operation $o_{j,i}$. For each first operation $o_{j,0}$ of job j we draw an arc from the source to the operation $(s, o_{j,0})$. Also for each last operation $o_{j,m-1}$ of job j an arc is drawn to the sink $(o_{j,m-1}, t)$. This arc is also weighted with the processing time of the operation.

Next the disjunctive edges are drawn. These edges model the machine constraints and are called the machine edges. The machine edges are initially undirected which gives them the ability to be directed in two disjunctive directions. For each machine we determine the set operations that are to be scheduled on this machine. For each set of operations we create a clique by interconnecting all the operations with edges.

In the process of solving the Job Shop Problem, the machine edges are being directed. When all disjunctive arcs are directed the result is a directed graph on which we can apply the longest path algorithm. By placing an orientation on the machine edges we fix the execution order

between the two operations. If the fixed machine edges (disjunctions) cause a cycle in the graph, an infeasible solution is obtained. In this case the precedence relations for a job are violated. Figure 1.1 gives an example of a disjunctive graph with undirected machine edges for a job shop problem with three machines and three jobs.

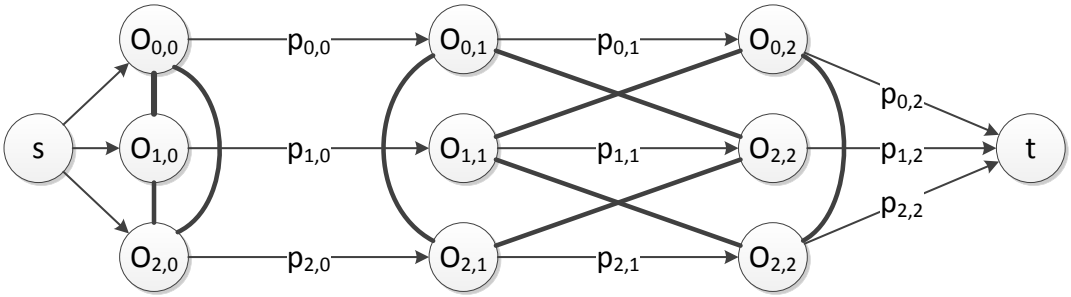


Figure 1.1: Example of a disjunctive graph in job orientation for a job shop problem with three jobs and three machines. The bold edges represent the machine edges. The arcs in between operations represent the job arcs.

When determining the makespan objective value we apply a longest path algorithm on the disjunctive graph. Since vertex t can only be visited after all job operations are visited, it determines the longest path of the schedule. Therefore it suffices to calculate the longest path from s to t.

MIP Formulation

Alternatively a Mixed Integer Program (MIP) can be formulated to model the optimization problem. There are several possibilities to model the decision variables and constraints, however we will only present one approach. The constraints in the following MIP formulation [26] are based on the conjunctive and disjunctive arcs from the disjunctive graph formulation:

$$\min \quad s_{n+1} \tag{1.2}$$

s.t.

$$s_{\sigma_i} \geq s_i + p_i \qquad \forall (i, \sigma_i) \in C \tag{1.3}$$

$$s_j \geq s_i + p_i - M(1 - x_{ij}) \qquad \forall (i,j) \in D \tag{1.4}$$

$$x_{ij} + x_{ji} = 1 \qquad \forall (i,j) \in D \tag{1.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in D \tag{1.6}$$

$$s_j \in \mathbb{R}^+ \qquad \forall j \in O \tag{1.7}$$

The objective function function 1.2 is similar to the sink vertex t from the disjunctive graph formulation. Set C in Constraint 1.3 consists of the conjunctive job arcs. In this set also the arcs from the final job operations to the sink vertex are included. This constraint forces the successor operation to start only when its predecessor operation is finished. Constraint 1.4 makes sure that operations on a machine do not overlap. Set D consists of all the disjunctive arcs from the

disjunctive graph formulation. By appropriately choosing a big number for M the constraint makes sure that overlaps cannot occur. In our case M is set to the sum of all processing times. In Equation 1.8 decisions variable x_{ij} effectively fixes the direction of the disjunctive edge between operation i and j:

$$x_{ij} = \begin{cases} 1 & \text{if operation } i \text{ is scheduled before operation } j \\ 0 & \text{otherwise} \end{cases} \tag{1.8}$$

Since the decision variable x_{ij} is defined for each disjunction, it means that this variable is mirrored. This is caused by the fact that each disjunction can be fixed either way. Constraint 1.5 makes sure that only one direction is chosen for each disjunction. Set O in Equation 1.7 consists of all operations and the dummy operation $n + 1$, which is equivalent to the sink node t in the disjunctive graph formulation. For each operation j in set O the start time variable s_j is defined.

By solving this formulation, a decision on the disjunctions will be found that leads to the minimum makespan. Although the amount of operations is limited, each disjunction requires two decision variables. Furthermore with each additional disjunction more constraints are added to the problem, leading to large problems for relative small instances. Consider for instance the job shop from Figure 1.1. This job shop consists of a total of 3 jobs and 3 machines; leading to 10 start time variables, 18 decision variables and 29 constraints. A job shop consisting of 4 jobs and 4 machines already has 17 start time variables, 48 decision variables and 65 constraints.

Since this formulation requires binary decision variables, a branch and bound algorithm often needs to be applied. Therefore solving the Standard Job Shop Problem by applying this formulation will already result in large computation times for moderate instances.

1.3 Local Search

Solving the Job Shop Scheduling Problem to optimality may take a large amount of computation time due to the fact that the problem is known to be NP-hard in the strong sense [15, 14]. Where for other NP-hard problems we are able to solve large instances, we find that the Job Shop Scheduling Problem stays notoriously hard to solve, even for small instances.

Common approaches to solve these problems are to apply Local Search heuristics, which will calculate an approximate solution or Branch and Bound algorithms. Both of these approaches require a good understanding of the underlying problem.

For the optimization of the Job Shop we have chosen to apply a *Simulated Annealing* [11, 4] local search technique. The overall process of the local search is expressed as a flowchart in Figure 1.2. The flowchart represents a local search framework where several key process are defined by elements to which a specific implementation can be given. The shaded elements in Figure 1.2 represent common functionalities. In this thesis we try to optimize the Job Shop Scheduling Problem by applying a local search algorithm together with a collection of new techniques. The new techniques consist of several implementations of local search components that are common throughout other local search approaches. By combining specific implementations of elements in the flowchart, the search process will be able to produce better schedules for the Job Shop Problem.

The standard implementation of the *Calculate makespan* process is done by applying the

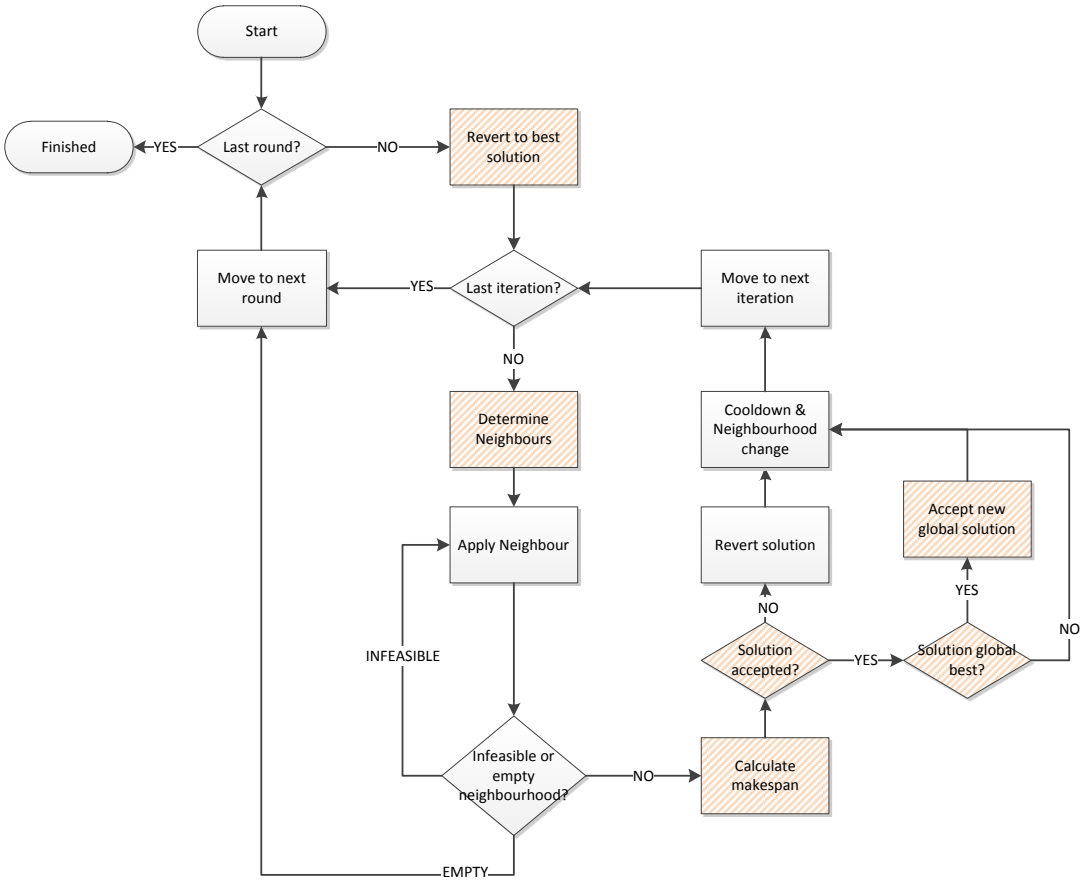


Figure 1.2: Flowchart of the local search architecture. The shaded elements represent common local search components to which a specific implementation can be given. By combining custom implementations of the components, new search approaches are built.

simulation, explained in Section 2.2.2. The actual search process is based on a two layer system. The outer loop, which runs over the amount of allowed restarts, (re)initializes the solution and resets the current temperature to the initial configured temperature. We call the outer iterations *rounds*. The local search component *'Revert to best solution'* is responsible for initializing a solution on which the search will be based. The actual implementation is to always initialize the search process with the global best solution found so far.

Furthermore the search process starts with an initial solution where the jobs are scheduled as a flowshop. For Job Shop Scheduling Problems this initialization is comparable to the worst case solution, since there is minimal mutual machine activity. However by giving all the machines the same job order we make sure that we start with a feasible schedule.

The inner loop is limited to a fixed number of iterations. These iterations are known as the common iterations in local search processes. After a fixed amount of iterations the temperature T is lowered by multiplying the temperature with a configured cool-down factor. This forces the local search to converge to better solutions.

The component *'Solution accepted?'* determines whether the neighbour solution is accepted as new current solution. In this process the objective value of the neighbour solution and the

current solution are compared. If the neighbour solution has a smaller objective value, the solution is always accepted. Otherwise the solution is only accepted with a certain probability. In the case the neighbour solution has a higher objective value, a random number R in the range $[0, 1]$ is generated and compared against the temperature T. If $R \leq T$ the neighbour solution is accepted, otherwise the neighbour is disregarded. The temperature variable T is used as a probability number for accepting worse solutions. This approach is similar to the simulated annealing schema used in [10].

The classical simulated annealing approach is to calculate the difference between the objective values, Δ, and determine $exp(\frac{-\Delta}{T})$ as the probability of acceptance. In this research we have chosen not to implement this classical approach of accepting worse solutions. The main reason for this decision is based on the fact that Δ is determined by the difference in objective value for the two solutions. Since we apply this local search algorithm to stochastic instances, Δ will also be a stochast. Therefore if we would apply the classical method, the behaviour of accepting solutions will be unstable. By using the temperature T as a probability number, we make sure we have a stable acceptance probability that is not influenced by stochastic realisations. Section 2.5.5 discusses the difference in performance for the chosen cool down approach compared to the classical cool down approach.

In the process *'Determine Neighbours'* a list of neighbour swaps is determined. The only swaps that are allowed correspond to a swap of two consecutive operations on a machine. When a swap is applied between two consecutive operations, the machine indexes of the operations are exchanged and thereby reversing the order of the operations. The swap of two consecutive operations can also be related to the disjunctive graph from Section 1.2.2 where the reversal of a disjunctive arc relates to a swap between the two operations.

In this thesis we have focussed on applying two different swap neighbourhoods:

- Critical path block swap
- Waiting left shift

As we can see in Figure 1.2 after each iteration a cooldown is applied and a check is done whether the neighbourhood must be changed. To facilitate the neighbourhood changes, a pool of neighbourhoods is created consisting of {Critical path block swap, Critical path block swap, Waiting left shift}. By using the Critical path block swap neighbourhood twice, we effectively weigh the neighbourhoods $\frac{2}{3}$ / $\frac{1}{3}$ in order of description. If in the process of applying neighbours it takes too long to find an improving solution, a neighbourhood change will be applied by moving to the next neighbourhood in the pool. As long as the neighbourhood consists of swaps that improve the solution, the neighbourhood is never changed. Furthermore if a neighbourhood is empty, the current neighbourhood is switched for the next neighbourhood. The swaps from the current neighbourhood are chosen randomly.

1.3.1 Criticalpath block swap

The critical block swap is based on adjusting the order of operations that belong to the critical path. The term critical path or critical chain originates from the project management theory. It is defined as the sequence of events that prevent the schedule from finishing earlier.

For a job shop the critical path determines the length of the schedule. Therefore the only way of decreasing the length of the schedule is by shortening the critical path. Previous research

results from Van Laarhoven et al. [27] have showed that the reversal of disjunctive arcs on the longest path will always lead to feasible solutions. Nowicki and Smutnicki [18] additionally showed that we can focus on reversing arcs of only the first pair and last pair of operations on a critical block. The Criticalpath block swap neighbourhood consist of the first and last pair of operations on the critical blocks.

1.3.2 Waiting left shift

When optimizing the Standard Job Shop Scheduling Problem, we see that in order to improve the objective value, waiting times need to be reduced. Bad schedules come from operations that cause a significant amount of waiting time on machines. By reducing these waiting times, the length of the schedule becomes shorter and the objective value improves. The idea of the Waiting left shift neighbourhood is to reduce the waiting time of the operation that causes the largest waiting time.

First the job consisting of the operation with the largest waiting time is obtained. Then for each operation in that job, a swap with the predecessor operation on the machine is constructed. This effectively allows all operations of that job to shift left.

If we look at how waiting times are introduced we see that a machine wants to start processing an operation that is not yet available. The only way to ensure that the operation becomes available earlier, is to make sure that the predecessor operation in the job is finished earlier. Since that operation also has a predecessor we have chosen to include left shifts for all the operations of that job.

The power of this neighbourhood lies in the fact that a combination of swaps is able to left shift larger parts of a job. Since this neighbourhood is not directly related to the critical path, this neighbourhood will not always find improvements. However, since the relative positions of jobs is changed, this neighbourhood is able to modify the structure in such a way that new solutions can be explored. In the case a local optimum is found, this neighbourhood can be employed to search for new regions in the solution space.

The Waiting left shift neighbourhood is not guaranteed to always produce an acyclic disjunctive graph. Therefore when swaps from this neighbourhood are applied, the solution has to be checked for feasibility.

Chapter 2

Stochastic Job Shop Scheduling

"Always be a little improbable."
— *Oscar Wilde*

2.1 Introduction

When optimizing scheduling problems, it is often the case that the actual input data of the model differs from the data that was used in optimizing the problem. To overcome this problem, the theory of robust scheduling has been developed. For robust scheduling must take into account various uncertainties that are inherent to the real-life models. In the context of Job Shop Scheduling we can identify the following list of examples that relate to uncertainties:

- Uncertainty regarding which jobs or operations will occur in a given scheduling problem. For example the set of activities can be unknown at forehand and will only become available in an online setting. In this case, an optimal decision can not be made in a deterministic way and sometimes decisions need to be changed when new online information becomes available.
- Uncertain properties of jobs and operations. For example actual processing times are stochastic or operation orders are unknown. These kinds of uncertainties limit the amount of methods that can be applied when optimizing such problems. Especially when the processing times are stochastic, it is hard to calculate a reliable objective value.
- Properties of machines can be uncertain, resulting in very unstable systems for which it is hard to apply deterministic solution approaches. As in the real world, machines can break down and thereby cause delays in the schedule.

Although the above list is limited, real-world applications often have one or more uncertainties from the above categories. When applying scheduling algorithms to instances with uncertainties, the actual outcome can have a high degree of disturbance. It is therefore important to apply methods that take these uncertainties into account. In this chapter we give an overview of different methods that can be employed to optimize the Stochastic Job Shop Scheduling. Our goal is to research new techniques that are able to produce Job Shop Schedules that are robust for specific uncertainties. The robustness of the schedules produced by the different methods is defined by the performance of schedule, when applied in a nondeterministic environment.

For the Job Shop Scheduling Problem in the deterministic setting, a lot of research already has been done, resulting in fast and simple methods that can produce near optimal schedules [19, 18]. To make the Job Shop Scheduling model applicable for real life applications we introduce stochastic elements. Our goal in this chapter is to develop Local Search components that facilitate the optimization of the Job Shop Scheduling Problem under the premise that operation processing times are stochastic. During this research the main question is "What combination of solution approaches performs best when optimizing the Stochastic Job Shop such that when we estimate the quality of the resulting schedule by a simulation, the mean makespan is minimal?".

In Section 2.2 we first give a formal definition of the Stochastic Job Shop Scheduling model, together with a definition of the simulation framework, used to calculate the objective function. Section 2.3 explains the different solution approaches we employed to optimize the Stochastic Job Shop. In Section 2.4 a description of the experimental setup is given. Section 2.5 analyzes the results of the experiments. Section 2.6 discusses the results and gives an overview of pointers for further research.

2.2 Formal definition of the Stochastic Job Shop Problem

In this chapter we mainly focus on the stochastic extension of the standard Job Shop Problem with unlimited buffer capacity. Although several components of the Job Shop may be subject to uncertainty, we only consider *stochastic processing times*.

Other applications include for example:

- Machine failures, where some machines will breakdown after an unknown amount of time and need time to be repaired.
- Uncertain changeovers, when some operations require setup times that are stochastic.
- Travel times, when jobs move to the next machine and the travel time is stochastic.

In the Stochastic Job Shop we introduce the stochastic processing time variable $P_{j,i}$. Processing times that remain deterministic will still use the $p_{j,i}$ processing time parameter, whereas stochastic operations will use the stochastic variable. When observing processing times, the $p_{j,i}$ variable will record the realization of the processing time. Note that for the deterministic operations, this variable is equivalent to the configured processing time. In the sections that follow we always assume that the underlying distributions of the stochastic processing times are known. Furthermore, we assume that the distributions are mutually independent.

Graham et al. [6] proposed a notation in 1979, that allows us to conveniently classify scheduling problems according to their properties. This notation, called the three-field notation scheme, consists of the three properties $\alpha \mid \beta \mid \gamma$, which indicate respectively the machine environment, the job characteristics and the optimality criterion. Since the original notation only covers deterministic instances, we have to introduce the notation of stochastic processing times and a stochastic optimality criterion:

$$J \mid \text{stoch. } p_j \mid E(C_{max}) \tag{2.1}$$

In Equation 2.1 J indicates that we are looking at a job shop problem. To indicate that operations can have stochastic processing times, the β field contains stoch. p_j. Furthermore due

to the fact that processing times are stochastic, all other properties related to the processing time are therefore also stochastic. In Equation 2.1 this is indicated by letting γ field contain an expectation criterion, in our case the *expected makespan*, which has to be minimized.

Throughout the remaining part of this chapter, the problem formulation is characterized by the definition given in Equation 2.1.

2.2.1 Objective Function

In this thesis we will only consider the *makespan* objective function. The makespan of a schedule is defined by the completion time of the last operation in the schedule, formally defined as C_{max}. According to the definition given in Subsection 1.2.1, this is equivalent to determining $max(e(Oj, i))$ over all operations Oj, i.

When a schedule is computed, the first operations of the jobs are released allowing machines to start processing. If for example machine m_1 first needs to process operation $O_{1,2}$, it first has to wait until operation $O_{1,1}$ is finished. In this situation the machine cannot start processing at the very beginning of the schedule, and remains "waiting". When all jobs and therefore all operations have finished processing, the Job Shop is completed.

In the deterministic case, the calculation of the *makespan* is done by applying a longest path algorithm to the disjunctive graph. A commonly applied algorithm is Dijkstra's algorithm [5] which runs in $O(|E| + |V|log|V|)$ time. Since we consider problems where the processing times are nondeterministic, the makespan is also a stochastic variable. Therefore the calculation of the objective function must also adhere to the rules of stochastic environments. The evaluation of the makespan is done by applying a discrete-event simulation.

2.2.2 Simulation model

For the Job Shop Problem with stochastic processing times, a simulation is required in order to calculate the *makespan*. This section describes the simulation model that is implemented to achieve the correct calculation.

For each machine in the Job Shop problem we can define a machine entity and give it his determined schedule. For every job in the instance, the first operation is initially marked Available. Whenever an operation $O_{j,i}$ of job j is finished, the next operation $O_{j,i+1}$ becomes available. This ensures that the precedence relations of the jobs are satisfied. Since each machine can only do one task at the time, the precedence relation of the machine schedules is also satisfied.

Since we assume that for the Stochastic Job Shop extension we have unlimited buffers, a machine can always start processing a new operation when its previous operation is finished. The previous operation on that machine goes to the unlimited buffer when it leaves the machines.

A Job Shop instance is defined by the following characteristics:

- A set of jobs with for each job a set of operations that must be executed in a given order. This information is fixed for the problem.

- A configuration of processing times. Each operation is assigned either a deterministic processing time or a stochastic processing time. Since we already made the assumption that the underlying probability distributions are known, we can attach a function to the processing time of an operation, that returns a random number from this distribution.

- A set of machine schedules, i.e. the order in which operations are processed on the machines.

For each operation in the Job Shop, the following states are applied in order:

- *Unavailable:* Each operation first starts with the state Unavailable. Whenever an operation is Unavailable it means that this operation still has predecessors in its job that need to be processed.
- *Available:* If all the predecessors of an operation have been processed, the current operation becomes available for processing. Therefore the state of this operation is set to Available. Note that for each job the first operation has no predecessors and therefore the first operation of each job is also set to Available.
- *Processing:* When a machine starts processing an operation, the state of that operation is also set to Processing.
- *Finished:* When the machine has finished processing an operation, that operation is marked Finished, and the successor operation within the current job is marked Available.

For the machine entities the following states are defined:

- *Idle:* If a machine has finished processing the last operation of its machine schedule, the machine is marked idle.
- *Waiting:* Whenever a machine continues to the next operation of its machine schedule and the next operation is not yet available (due to the precedence relation of the jobs), this machine should wait until the operation becomes available. During this time the machine is marked Waiting.
- *Processing:* When a machine is processing its current operation, the machine is marked Processing during the length of the operation's processing time.

The control flow of the machine after processing operation $O_{j,i}$ of job j consists of the following possibilities:

1. Job j is finished and therefore removed from the simulation.
2. Job j is placed in the unlimited buffer of $machine(O_{j,i+1})$.

```
method OperationProcessed()
{
        CurrentOperation <- Finished;

        if (CurrentOperation is LastOperationInJob)
        {
                ContinueMachineSchedule();
                return;
        }
```

```
        CurrentOperation.Next <- Available;
        Place CurrentOperation.Next in Buffer;
        ContinueMachineSchedule();
}
```

Next the machine will try to process the next operation defined in its schedule. The following list of possibilities describe the process if operation $O_{j,i}$ of job j is the next operation in the machine schedule:

1. If operation $O_{j,i}$ is *Available* (either because it is the first operation of a job, or the operation resides in a buffer) the machine can start processing.

2. If operation $O_{j,i}$ is not available the status of the machine is set to *Waiting* for operation $O_{j,i-1}$ to be finished. Eventually since there is no deadlock in the machine schedules, operation $O_{j,i-1}$ will be finished and thereby marking operation $O_{j,i}$ *Available*.

```
method void ContinueMachineSchedule()
{
        CurrentOperation <- GetNextOperationInSchedule();

        if (CurrentOperation is NULL)
        {
                MachineState <- Idle;
                return;
        }

        if (CurrentOperation is Available)
        {
                MachineState <- Processing;
                Schedule OperationProcessed;
                return;
        }

        MachineState <- Waiting;
}
```

When a machine has finished processing the last operation of his machine schedule, the state of the machine is set to *Idle*. In the end when all machine states are *Idle* the simulation is finished and the completion time of the last operation determines the *makespan* of the instance. Figure 2.1 gives an overview of the events that implement the above described behaviour.

When the simulation is completed, all jobs are scheduled and finished. The timestamp of the last *Operation Processed* event is therefore also the makespan. Since the simulation records for each operation its starting and completion time, respectively $s(O_{j,i})$ and $e(O_{j,i})$, we also have a method to determine the critical path.

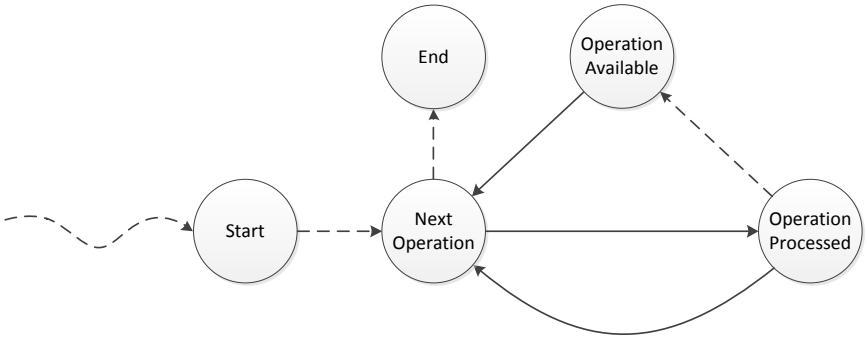


Figure 2.1: Event graph of the simulation entities. The notation is taken from Averill M. Law (2007) [12]. Dashed arcs represent events that are scheduled immediately, solid arcs represent events that are schedule in the future.

2.3 Solution Approach

One of the common approaches in the literature for solving the Job Shop Problem is to apply a local search heuristic. However a local search heuristic is not directly applicable to stochastic problems since the stochastic aspects introduce various complexities. One of the complexities is for example the calculation of the objective value. Since the objective value consists of stochastic processing times, the value itself is also stochastic. When comparing two objective values we need to take into account we are comparing two realizations of stochastic variables. The main aspect of this part of the thesis is therefore the development of several methods that facilitate the search process for stochastic instances. Each of the methods consists of implementations of several components of the local search framework introduced in section 1.3.

This section gives an overview and implementation details of the methods we designed for solving the Stochastic Job Shop. The development of methods is divided in two parts. First the development of classical methods is introduced, that work directly on the given probability distributions to estimate deterministic processing times. These methods apply a derived amount of slack to the processing time variable p_j, related to the underlying probability distribution. Next an overview of more sophisticated methods is given. These methods require more computing power and are applied in several components of the local search framework.

2.3.1 Classical Methods

Most of the common applications that try to cope with uncertainties attempt to make the schedule more robust. One of the classical methods that is employed in robust scheduling, is to find a deterministic value for a stochastic variable. The underlying probability distribution can either be known or unknown.

Next, during the optimization, the derived deterministic value is used instead of the stochastic variable. Usually the derived value is chosen in such a way that during the search process a small amount of 'slack' is reserved. This is done by making sure that the derived value is (slightly) larger than the mean of the underlying distribution. Considering values smaller than the distribution's mean would be too opportunistic.

The quality of the solutions that are produced by these classical methods highly depend on whether the correct amount of slack is reserved. A successful solution will reserve just enough slack to cope with a reasonable amount of disturbances. The derived value must also somehow resemble the underlying distribution's shape to be effective.

In general, the process of deriving deterministic values is captured by the following equation:

$$\tilde{p}_j = \mathcal{F}(P_j) \tag{2.2}$$

In the derivation given by Equation 2.2 a function has been applied to the stochastic variable P_j. The derived value $\tilde{p}_j$ is then used by the local search to optimize the problem. In the following subsections two different methods of implementing the derivations are proposed.

Robustness Factor

The most simple method for deriving a deterministic value from a stochastic variable is to apply a Robustness Factor. When solving nondeterministic problems using this approach, the stochastic entity, which is in our case the processing time, is multiplied by this factor. This is effectively the same as multiplying the mean value of the stochastic variable, since the configured processing times are defined as the mean of probability distribution.

Suppose we have a stochastic variable P for which we know the underlying distribution. First we take the distribution's mean value $\overline{p}$. If the underlying distribution is not known, we estimate it by taking n observations p_i $(i = 1 \ldots n)$ and apply equation 2.3.

$$\overline{p} = \frac{1}{n} \sum_{i=1}^{n} p_i \tag{2.3}$$

Next we apply the Factor Robustness approach by multiplying the mean with a factor. Equation 2.4 gives an application of the Robustness Factor 1.2 for distributions P with mean $\overline{p}$; F indicates the chosen factor:

$$\begin{aligned} F &= 1.2 \\ \tilde{p} &= \overline{p} \cdot F \end{aligned} \tag{2.4}$$

The final step in this approach is to apply the local search algorithm with the calculated value $\tilde{p}$ as indication for the processing time. The Robustness Factor of Equation 2.4 gives an example of how 20% extra time relative to the average processing time $\overline{p}$ is taken into account during the local search. The actual factor does not have a real relation to the problem, in fact when applying this approach to a problem one must first determine the correct factor. Determining the factor that produces good and robust solutions is on itself a heuristic process and is therefore not trivial to apply to real life scenarios. Also keep in mind that when using a sampled or derived average, the underlying distribution's shape plays an important role in the actual robustness of the solution. Therefore using only the mean value might not give us enough information for deriving a robust deterministic processing time.

Robustness Percentile

A similar approach to obtain robust solutions, is to apply Percentile Robustness. Note that we assume that the type and parameters of the underlying distribution are known. In the case the

underlying distribution is unknown, sample data should be obtained to estimate the probability distribution.

By having access to this information we are able to calculate percentiles of the distribution. In the application of Percentile Robustness, the properties and therefore the deviation as well as the distribution's characteristics are being preserved. Take for example an Uniform distribution and an Exponential distribution with the same mean value. When applying Factor Robustness, the two derived processing times will be equal. In the case of Percentile Robustness the derived processing time will receive a larger value for the Exponential distribution. This way the large tail of the Exponential distribution, and therefore the large deviation, is taken into account.

In statistics, the 60th percentile x from probability distribution P indicates the boundary x for which 60% of the observations from P are smaller than x. If we take the Robustness Percentile as derivation for the processing time, we effectively make sure that a given percentage of the observations are smaller than the value which we used in the search process. Similarly, if we would for example take percentile 60, the robustness comes from the fact that 60% of the observations will be optimistic compared to the value we have derived.

When comparing the Percentile Robustness to the Factor Robustness we can find that there are some similarities. In fact it is possible for a limited set of probability distributions to derive a percentile from a given factor and vice versa. To show how this can be done we first look into the process of calculating a percentile from a given probability distributions.

In statistics the Cumulative Probability Distribution $F(x)$ calculates the probability y that a random variable will be found at a value less than or equal to x. For some distributions the Cumulative Probability Distribution has a closed form and have therefore the ability to be inverted. This is called the Inverse Transform Method [20]. The equation derived in 2.5, $-\beta \cdot ln(1-y)$, can be used to calculate a number x from the Exponential distribution such that $(y \cdot 100)\%$ of the observations are smaller than or equal to x. Equation 2.5 shows how the Exponential distribution with arrival rate parameter β is inverted. For the Exponential distribution, parameter β is equal to the mean value of the distribution.

$$\begin{aligned} F(x) &= y \\ F(x) &= 1 - e^{\frac{-x}{\beta}} \\ y &= 1 - e^{\frac{-x}{\beta}} \\ 1 - y &= e^{\frac{-x}{\beta}} \\ ln(1-y) &= \frac{-x}{\beta} \\ -\beta \cdot ln(1-y) &= x \end{aligned} \tag{2.5}$$

By looking at the derived equation we see that the percentile value of the distribution can be expressed by the multiplication of β and a function application of the percentile. As example we calculate the 70th percentile of the Exponential distribution:

$$-\beta \cdot ln(1-0.7) = \beta \cdot 1.2 \tag{2.6}$$

In Equation 2.6 we see that percentile 70 of the Exponential distribution relates to a factor 1.2 for the Factor Robustness. By using this technique we can invert all given percentiles of the Exponential distribution to a factor relative to the distributions mean β.

Still if we have different distributions, the percentiles that we apply do not correspond to the same factors. Take for example the Exponential distribution again. When we want to use the percentile that corresponds to the distribution's mean, we require $ln(1-y)$ to be -1 implying that $y = 1 - e^{-1}$. In this case y corresponds to percentile 0.63. If we would have chosen a percentile smaller than 0.63 we would have considered a number lower than the distribution's mean and therefore the local search would be too optimistic. The application of percentiles must therefore be carefully checked against the underlying distributions.

Note that the conversion between percentile and factor does not hold for all probability distributions. Furthermore, different distributions have different properties, therefore if we consider two non similar distributions, the percentile properties can not be directly related. This effectively limits us the ways in which we can apply this classical method.

2.3.2 Sophisticated Methods

To overcome the limitations of the straightforward classical methods, we have developed more sophisticated methods. For each new method, specific implementations of components of the general Local Search architecture from Section 1.3 are given. The following subsections describe the details of how several components of the new methods are implemented. Central in this development is the idea that replacing the stochastic variables with deterministic derivations, like in the classical methods, does not produce robust schedules. By introducing a simulation as a replacement for the deterministic objective function, with the support of the sophisticated methods, we try to find schedules that have a better average makespan.

Processing Time Sampling

The processing time sampling method is very similar to the idea of applying the local search algorithm, in which we replace the stochastic processing times with their probability distribution's mean. When applying the processing time sampling method, local search will use the average processing time from a fixed amount of samples. The difference with processing time sampling instead of applying deterministic processing times, is the fact that when sampling is applied, small amounts of variation occur.

Whenever the simulation needs to determine the length of an operation, this method will calculate the average of the processing time by taking 100 samples from the distribution. Note that when we use the processing time sampling method, we are actually approximating the underlying probability distribution's mean. The concept of the application of processing time sampling is that we will carefully try to let the local search notice some of the disturbances of the underlying distributions. Because this application is based on sampling, it will always have disturbances to some extent.

This method can be implemented in two ways. The decision here to make is whether we only apply the processing time sampling method once, and use the calculated average throughout the whole local search, or we apply this method each individual time whenever the length of some operation's processing time needs to be determined. We have chosen to use the latter implementation. An obvious reason for this decision is the fact that the more often we calculate sampled processing times, the more information the local search receives.

Result sampling

If we only consider the sampling of all the individual processing times there is a possibility we only allow the local search to see average based results. This is due to the fact that all processing times are averaged. In practice, only a portion of the operations will receive an average processing time while other operations will have smaller or larger processing times. Taking average processing times for all operations will remove this interaction. It is our goal to take other aspects such as uncertainty, e.g. variance, into account, therefore the Processing Time Sampling application may not behave as expected. In this thesis we therefore have tried to construct a method that, in the application of a standard local search framework, will find robust solutions. The simplest approach to this idea was to base the calculation of the *makespan* on the results of a single simulation run of the schedule.

In the application of *single result sampling* the local search will see just one realisation of the schedule. The schedule is calculated by applying a simulation and taking only a single sample for each processing time variable. Since all operations in the schedule are influenced by stochastic processing times the makespan can behave very erratic. This behaviour allows us to make decisions based on situations that could occur in real life. However a large disadvantage of the *single result sampling* method is that no two realisations will look the same. Even in different iterations of the local search, a schedule that first might have been a good schedule, can turn out to be very bad in later iterations.

To limit this effect we extended the heuristic to *five times result sampling* and *ten times result sampling*. Instead of running the simulation once, the simulation will be executed either with five or ten runs, depending on the heuristic. The results of executing the schedule are then averaged. Again in each individual simulation run only one sample for each processing time is obtained. Finally average makespan over the simulated schedules is used.

This technique behaves inherently different compared to the processing time sampling heuristic since with the processing time sampling all operation lengths are averaged. With result sampling there is still the possibility that processing times have either optimistic or pessimistic realisations. Furthermore it is the collaboration of operations that in the end determine how a schedule performs. Since each individual simulation obtains one sample from each processing time variable, this interaction is preserved.

Sample based neighbourhood repository

When only having one sample for which to determine the neighbourhood repository, the decisions are simple. There is only one schedule from which we can identify the *Critical block swaps* and the *Waiting left shifts*. If we consider the multiple times result sampling from Section 2.3.2, we can not just base our neighbourhood repository on only one of these results. In fact if we would base our neighbourhood on only one of the result samples, the local search would have a hard time converging.

To support the search process in case of multiple times result sampling, we have decided to implement a heuristic that fills the neighbourhood repository with neighbours from all the samples. For each simulation result from the multiple times result sampling we determine the *Critical block swaps* and the *Waiting left shifts* that are eligible. The swaps and shifts are called neighbours since when applied to a schedule, a neighbour solutions is obtained.

Figure 2.2 gives a schematic overview of how the neighbourhood repository is filled with

neighbours from individual results. The application of the sample based neighbourhood repository also gives us another advantage. Consider for instance a schedule with several stochastic operations. After five result samples have been calculated you notice that three result samples return the same subset of neighbours. Since the neighbourhood repository randomly chooses neighbours from its collection, it means that neighbours that occur more often have a higher possibility to be chosen. This concept is in fact very similar to neighbours that have different weights associated with them. A neighbour with a higher weight is considered to be a better candidate.

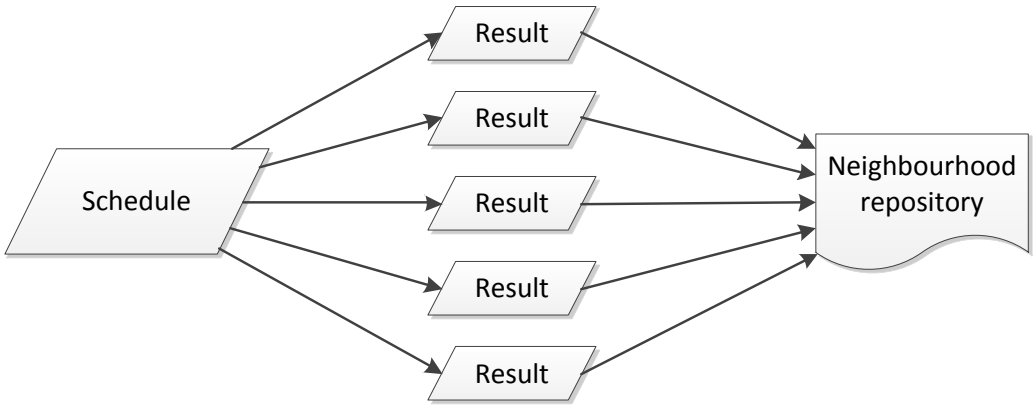


Figure 2.2: Example of sample based neighbourhood repository. The *Schedule* is the result of a local search iteration. This *Schedule* is then simulated five times. The neighbourhood repository is filled with neighbours from each individual simulation result.

Sample Dominance Test

The process of determining whether one schedule is better than the other requires a method that can indicate the relative performance of the schedules in terms of the objective function. This we call the *dominance* of the schedules. In our case the objective function consists of the makespan, which is the critical path of the schedule; so the dominance of the schedules is based on this performance measure. We already know that the makespan is a stochastic variable, so therefore the dominance between two makespans needs to be determined.

If we would just simulate two different schedules and compare their makespans, our indication of the dominance would be biased. This is due to the fact that in the process of determining the makespans, the realizations of the processing times differ. Section 2.6.2 explains why this approach leads to suboptimal local search results. In order to apply a correct dominance test for the comparison in the local search, we have developed an additional test called the *Sample Dominance Test*.

When the current schedule is being related to the current global best solution, the *Sample Dominance Test* is executed. For both the schedules a fixed amount of results are sampled. In each iteration of calculating the result samples, the simulation of both schedules is based on the same random numbers, effectively applying *Common Random Numbers* [12]. After all samples

have been calculated the total of the makespans are calculated. Finally the sum of makespans is compared to each other, stating that the schedule with the smaller total makespan is dominant over the other schedule. In the local search the *Sample Dominance Test* determines whether the current solution is accepted as new global best solution.

Cutoff Sampling

When trying to optimize schedules with stochastic operations that have a large variance, an additional technique can be applied to stabilize the local search. If we take for example the Exponential distribution, it is easy to see that it is nearly impossible to be robust against realizations that range from percentile 95 up to percentile 100. Also realizations that are far under the mean of the distribution can be disregarded since these operations will just introduce waiting time on the machine and make the critical path switch to other machines. However we can never draw conclusions from sample based results since the *Lucky Ticket Theory* from Section 2.6.2 still plays large role, we will still try to disregard the region of samples that we do not want to consider.

When calculating the results of a single schedule, instead of considering all the samples found by the *Result sampling* technique from Section 2.3.2, we only consider a small subset of those samples. First we calculate for each sample the makespan for the given schedule, after which we sort the samples with respect to the makespan. Then we disregard a top and bottom percentage of those samples. Afterwards the final makespan estimator is based only on the fraction that is left. Since we wish to find schedules that are robust against processing times that are above the underlying distribution's mean, the fraction that we disregard from the top is smaller than the fraction disregarded on the bottom.

When comparing the actual performance of the different heuristics, we also need to take the Cutoff sampling into consideration. This means that next to comparing the heuristics based on an average that is simulated, the results also need to be compared based on only a fraction that is equal to the fraction used in the optimization process.

2.4 Experimental setup

In order to determine which method leads to good solutions, we need to define a generic experiment that can give us more insight. Important in this setup is that for each of the experiments the local search settings are kept the same. Therefore we decided that for each of the instance the following settings are preserved:

- Maximum of 10 restarts
- Maximum of 25000 iterations per restart
- Exponential cool down schedule with start probability 0.8 and cool down factor 0.95
- Cool down applied after 400 iterations
- Alternating neighbourhoods (Critical block swaps / Critical block swaps / Waiting left shifts) that change after 15 successive acceptance failures

Furthermore, each specific experiment is repeated 5 times, resulting in 5 possibly different solutions to the problem instance. The instances we used in our experiments are randomly generated since there are no benchmarks instances known for the Stochastic Job Shop Scheduling in the literature.

2.4.1 Generation of benchmark instances

This section gives an overview of the different instances that are used to determine the performance of the heuristics. Note that there is no commonly accepted benchmark repository for the Stochastic Job Shop, therefore we have decided to generate a new set of instances.

For each instance we first determine the number of machines and the number of jobs. By assumption and as a consequence, the number of operations per job equals the number of machines. Then by applying *factoradic base permutations* (Lehmer code, [23]) we can effectively shuffle the order of the operations so that each job will most probably have different paths trough the machines. In this process each job j will consist of operations $\{O_{j,i} | i = 0 \dots M-1\}$. Next by drawing a random number from $0 \dots (M-1)!$ and mapping that number to the factoradic base, we convert that number to a permutation of the numbers $0 \dots M-1$ [24]. Finally for each operation we store the found machine by defining the corresponding $machine(O_{j,i})$ index.

Now to turn the instance in a stochastic instance we record the processing times. For each operation we first determine the *base processing times*. This is done by drawing a random number that comes from a uniform distribution defined by $U(10, 20)$. The recorded *base processing times* determine the processing time for deterministic operations. Finally for some jobs or machines the *base processing times* for the operations are altered to form a specific distribution. The configured *base processing time* is used as a parameter to this distribution. In the process of turning deterministic operations into stochastic operations, first the subset of operations which will be stochastic is determined. The determination of operation subsets is done based on either complete jobs or complete machines. For example, if a job is chosen to be stochastic, all the operations will receive a stochastic processing time.

Next we decide what distribution to use and apply that to this subset of operations. For example an operation with a base processing time of 15 units can be altered to a stochastic processing time which follows the Exponential distribution given by $Exp(15)$. Another possibility is to alter the operation to form a Uniform distribution which will have a range of 40%. The underlying distribution is then given by $U(12, 18)$. Distributions with more parameters can also be modelled. By fixing one of the parameters we could also create a 4-Erlang distribution. The operation from the example would then have an underlying distribution given by 4-Erlang$(\frac{15}{4})$ = Gamma$(4, \frac{15}{4})$.

2.4.2 Benchmarking architecture

In order to have a standardized method of benchmarking the several approaches a benchmarking architecture is built. In this benchmarking architecture the Robustness Factor, Robustness Percentile, Processing Time Sampling, Result Sampling, Sample based neighbourhood repository, Sample Dominance Test and Cutoff Sampling solution approaches will be applied to the same instance in a way that all the heuristics get an equal and fair amount of computing power. We also made sure that the final analysis of the schedule, that is found by each experiment, is done in the same way for all results.

The process of benchmarking starts by first applying each solution approach five times. In this process each individual run will return a schedule that is the solution for the experiment. Finally, to calculate the performance of that schedule, we will calculate the objective function 1000 times by means of a simulation. The simulation uses the original processing time probability distributions to determine the real performance.

For an experiment, the input of the local search phase is defined by the following three properties:

- Stochastic Job Shop instance
  The instance file defines the number of machines, the jobs and the operations that belong to the job. For each of the operations the *base processing time* is given together with the machine index based on $machine(O_{j,i})$. The order in which the operations are defined explicitly defines the order in which the operations need to be processed.

- Local search settings
  The methods that are defined for each experiment have a specific configuration for the local search. Therefore each different type of experiment requires a different local search definition. The settings stored in the local search settings file configure the selection of solution approaches that are required for that experiment.

- Disturbance settings
  If an instance has operations with stochastic processing times we record the disturbances in this settings file. For each of the operations where a stochastic processing time is applicable, we record the disturbance of that operation. This way we are able to model the disturbances in a different file. The *base disturbances* files record the disturbances for all operations that have stochastic processing times. The classical methods that require deterministic processing times will have individual disturbance configurations, each recording their specific derived processing time.

The local search phase is then followed by the objective function phase. In this phase each of the schedules that were found by the local search phase are simulated 1000 times. For each of the schedules the simulation is set up with the following properties:

- Stochastic Job Shop instance
  This is the same file used in the application of the experiment.

- Simulation settings
  The settings for running the simulations are fixed. The total amount is set to 1000.

- Schedule
  This is the schedule that is found by the local search phase.

- Base Disturbance settings
  When evaluating the schedule found by a heuristic process, we need to plug in the disturbance settings file that contains the original probability distributions, to check what the performance is in reality.

Figure 2.3 gives a schematic overview of how a single experiment is benchmarked. The final outcome, after applying the benchmark architecture to the selected solution approaches, is

for each approach a list of 5 schedules and for each schedule 1000 simulation results found by applying the real disturbances. Next we need a way of comparing the different results. First, the 1000 simulation results of the five schedules are combined, resulting in a dataset of 5000 makespans. Finally for each set of 5000 results we calculate the following list of statistics:

- Average makespan
- Standard deviation of the makespan
- Minimum makespan
- Maximum makespan
- Average makespan based on cutoff sampling
  Since some of the local search heuristics are based on cutoff sampling, we will also take this performance measure in consideration. By selecting a subset of the 1000 simulation results based on discarding the top x_1 and bottom x_2 fraction of the results, the cutoff samples will behave very much the same as the objective function in the local search phase.

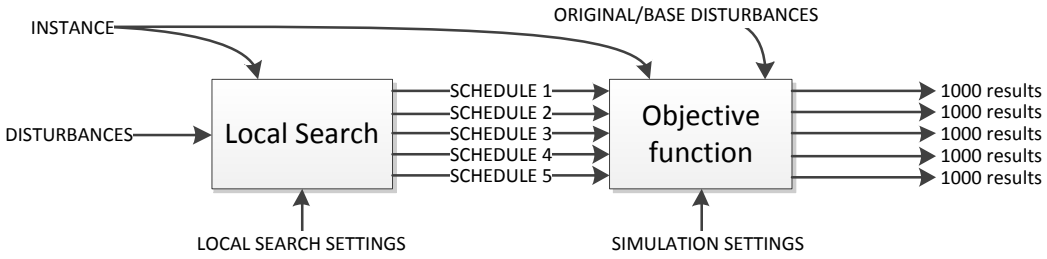


Figure 2.3: Schematic overview of the execution of one experiment.

The above statistics give us a good overview of the performance for a specific combination of methods. However, since the above list of statistics is very cumbersome, in the results we only include the average makespan. From the calculated makespan we also give the relative performance for the different methods. To create an accurate relative performance ratio, we take the smallest found average makespan value as 100%. Next for each result we calculate the relative performance by means of the following formula:

$$\frac{makespan}{makespan_{best}} \cdot 100\%$$

The application of a more sophisticated statistical analysis method proves to be very hard. This problem is caused by the fact that the makespan, that is actually a stochastical variable, does not follow a distribution we know. Next, since we only calculate five schedules, we can not do statistical analysis among those schedules because a sample count is too small to draw conclusions from. This leaves us with the relative performance analysis on only the calculated averages and deviations.

If the local search is able to produce robust solutions we anticipate on a series of observations to occur. First of all, if the local search is robust, we think that the average makespan over the 5000 simulations will be lower than the average makespan of other methods. Next to the fact that the average makespan should perform better, we also think that the standard deviation of such schedules is better. Robust schedules in this case can be expressed as schedules that have a *better average performance* and a *smaller deviation* between the simulated results of individual schedules, found by the same combination of solution approaches.

2.4.3 Generated Instances

For the analysis of the selected heuristics we made a broad range of different instances. For the initial experiments we have chosen to only focus on instances that consist of 10 machines and 10 jobs, resulting in instances that have 100 operations. The additional experiment sets vary on the amount of machines and operations. Furthermore we assume that all the jobs must visit each machine exactly once, which makes each instance effectively square.

For each operation we generate a random processing time coming from a uniform distribution characterized by a lower bound of 10 and an upper bound of 20 units. The additional experiment sets also vary on the upper bound of processing time, allowing operations to have a much larger processing time. As already explained, some of the operations will receive a disturbance, turning the instance into a stochastic instance. In our research on robust optimizing the Stochastic Job Shop we have focussed only on the following distributions:

- Uniform distribution defined by $U(0.8 \times p_{j,i}, 1.2 \times p_{j,i})$
- Exponential distribution defined by $Exp(p_{j,i})$
- 4-Erlang distribution defined by $Gamma(4, \frac{p_{j,i}}{4})$
- LogNormal distribution defined by $LogNormal(ln(p_{j,i}) - \frac{ln(2)}{2}, ln(2))$
- Deterministic defined by $p_{j,i}$

Note that some of the parameters have an adjusted processing time variable. The reason why the parameters are adjusted is because we want the mean of the distribution to be exactly the same as the base processing time, the parameter $p_{j,i}$.

Looking at the 4-Erlang distribution or the equivalent Gamma distribution with scale parameter 4, we know that the distributions mean is defined by:

$$Mean(x) = \alpha\beta \tag{2.7}$$

Since the scale parameter α is fixed to 4, we need to scale the β parameter, which is defined by the base processing time, down by 4. The altered distribution will have a variation defined by:

$$Var(x) = \alpha\beta^2 = 4(\frac{p_{j,i}}{4})^2 = \frac{p_{j,i}^2}{4} \tag{2.8}$$

Which is exactly four times smaller than the variation defined by the exponential distribution.

For the LogNormal distributions a similar transformation has been applied. The mean and variance of the LogNormal distribution are defined by the following formulas:

$$\begin{aligned} Mean(x) &= e^{\mu+\frac{\sigma^2}{2}} \\ Var(x) &= e^{2\mu+\sigma^2}(e^{\sigma^2}-1) \end{aligned} \tag{2.9}$$

Solving the formula for the mean by μ we ge the following Equation:

$$\begin{aligned} e^{\mu+\frac{\sigma^2}{2}} &= p_{j,i} \\ \mu + \frac{\sigma^2}{2} &= ln(p_{j,i}) \\ \mu &= ln(p_{j,i}) - \frac{\sigma^2}{2} \end{aligned} \tag{2.10}$$

Similarly we want to relate the variance of the LogNormal distribution to the Exponential distribution. For the Exponential distribution, the variance equals its mean squared. Since the mean parameter of the distribution is fixed to $p_{j,i}$, we can appropriately choose σ:

$$\begin{aligned} e^{2\mu+\sigma^2}(e^{\sigma^2}-1) &= \text{Mean}^2 = p_{j,i}^2 \\ p_{j,i}^2(e^{\sigma^2}-1) &= 1 \cdot p_{j,i}^2 \\ e^{\sigma^2}-1 &= 1 \\ \sigma^2 &= ln(2) \end{aligned} \tag{2.11}$$

Finally we can substitute σ^2 for $ln(2)$ resulting in a LogNormal distributions with the same mean and variance as the Exponential distribution:

$$LogNormal(\mu, \sigma^2) = LogNormal(ln(p_{j,i}) - \frac{ln(2)}{2}, ln(2)) \tag{2.12}$$

Disturbance sets

Since we assume not all operations for all experiments are stochastic, a subset of operations need to be selected which we turn into stochastic operations. The subset of operations is chosen by either selecting a set of operations or a set of machines, for which all operations in that job or on that machine will be subject to uncertainty. For the research to be as complete as possible we have chosen to create instances with four different sets of stochastic operations:

A 30% of the jobs / machines are stochastic

B 50% of the jobs / machines are stochastic

C 70% of the jobs / machines are stochastic

X 100% of the operations are stochastic

One of the difficulties in comparing the different methods and combinations of stochastic operation sets, is that the local search can result in schedules that are hardly comparable. This is due to the fact that we want to base our conclusions on more than one instance while instances are mutually not comparable since their underlying structure is inherently different. To overcome

this problem we have fixed the amount of distinctive instances, allowing some instance to be reused for other experiments. Looking closely at the properties we have described so far, we can see that the total amount of instance and disturbance combinations equals $4 \times 7 = 28$. In this equation 4 stands for the combination of available probability distributions and 7 stands for the combination of disturbance sets (3 jobs sets, 3 machine sets and 1 total set). The 7 disturbance sets all have a specific instance so that an experiment with job disturbance set A will always have the same job shop instance. For example the Exponential distribution on job set A has the same underlying instance as the Uniform distribution on job set A. By fixing the instances in this way we effectively apply the same instance for all disturbances, over the same set of stochastic operations.

Table 2.1 on page 36 gives an overview of the instances that are generated.

2.5 Results and Analysis

In this section we present the results and analysis of the experiments described in the previous sections. Before combining several solution approaches we will first start with an experimental setup to test what solution approaches perform relatively good. After we have found several method combinations that perform good on the experimental instances, we will apply these method combinations to a larger set of instances. The larger set of instances will give us insight in how the methods perform over a variable amount of disturbances and different probability distributions. As last experiment we will apply the best found methods to instances that are stretched in term of Job Shop size and processing time ranges. Finally this section is concluded with an analysis of the solution approaches in which we present the insights gained by the methods.

2.5.1 Determination of Methods

In the development of the final methods we first have multiple phases in which we adopt the new techniques. Since this process is important for the final outcome of methods, we present in this section some of the improvements we made to the methods.

Experiment set one: classical heuristics and single sampling

The main goal of our research is to find a robust optimization technique that makes use of sampling techniques. Since the most common technique to create robust schedules is to apply some slack time to processing times, we will start by applying the *Robustness Factor* for factor 1.2 and the *Robustness Percentile* for percentiles 60, 70, 80 and 90. Note that the factor and percentiles are chosen arbitrarily. Next to these heuristics we will also apply the *Single Sampling Method*. The characteristics of these heuristics consist of a single sample per objective function calculation and a neighbourhood built from this single sample.

The results of the above experiments are displayed in Table 2.2 on page 37. For each heuristic five schedules have been calculated by the local search. Next, each schedule has been simulated 1000 times from which an average makespan and a standard deviation have been obtained. The individual results have been sorted in an increasing order based on the average makespan. Next the methods are sorted according to the average over averages, displayed in the bottom line.

The best performing method is listed in the first columns and the worst performing method is listed in the last column.

From Table 2.2 we can clearly see that the classical robustness techniques perform better than the sampling technique, since the averages are smaller and the deviations are smaller. A strange effect that was encountered during this initial experiment was that during the local search of the *Single Sampling Method* the objective function obtained better results than the simulation runs afterwards. As we will later see, this side effect plays an important role in local search based on sampling techniques.

Experiment set two: multiple times sampling

To continue the method development we have considered another set of experiments to increase the performance of the sampling heuristics. In the following set of methods we have considered the relative performance of *Single Sampling*, *Five Times Sampling* and *Ten Times Sampling*. During the local search of the *Multiple Times Sampling*, the neighbourhood was based on the swaps found by the first calculated sample. The choice to base swaps on the first found schedule was made arbitrarily. Furthermore, the acceptance of better solutions was solely based on comparing the averages of the results. Also the *Processing Times Sampling Method* is employed in this experiment, in order to check whether it is successful to base the local search on processing times that come from sampling the underlying distribution 100 times. Table 2.3 gives an overview of the second set of experiments.

The results of the second set of experiments clearly show that the more we sample in the local search, the better average makespan is when simulating against the real distribution. Still the *Percentile Heuristic* performs better than the sampling techniques. If we compare the run data of the percentile heuristic with the sampling heuristics we see that it takes a fair amount of time for the sampling techniques to find optimal solutions. Especially the *Multiple Times Sampling* techniques have problems finding swaps that lead to better solutions. A reason could be that the swaps are not reliable enough since they are only based on one sample. Neighbours from samples in consecutive iterations are based on totally different realizations of the processing times and can therefore result in swaps that counter the swaps in earlier iterations. Therefore basing the swaps on only one sample may result in a search process that is not able to converge enough.

Regarding the *100 Processing Times Sampling*, one might think that the results will be robuster than when we apply the classical robustness techniques. This is however not the case. Since each stochastic operation operation is averaged, the so said 'cooperation' of processing times is flattened out. Furthermore, the objective function is less sensitive for disturbances and can therefore not reproduce the effects we want for creating robust schedules. Another way to look at this heuristic is to compare it with planning the operations on the mean property of the underlying distribution.

Experiment set three: neighbourhood repository based on all samples

The next improvement for the *Multiple Times Sampling Heuristic* is to fill the neighbourhood with swaps from all the individual samples. Swaps that can be obtained from more schedules can be regarded as better swap and should therefore have a higher probability to be chosen. By filling the neighbourhood repository with swaps from all the sampled results we effectively make

sure that better swaps occur more often.

The results of set three are shown in Table 2.4. Although the local search was able to find solutions earlier in the local search process, we see that in the end the robustness is not better or even worse. Even now we observe during the local search process, evaluations of the objective function that are better than the best results found by the classical heuristics. Combining these observations we must conclude that the search process is not able to capture representative disturbances.

Experiment set four: sample based, dominance test and cutoff sampling

The final experiment set in determining the best combinations of heuristics is based on the fact that sample based heuristics are sensitive for strange realizations of the underlying disturbances.

To overcome this problem a dominance test is employed, the check whether the current solution is better than the global best solution. The details of this heuristic are explained in Section 2.3.2. The application of this heuristic takes place where the acceptance test for a new global best solution is executed. Both the current solution and the global best solution will be feeded with the same set of underlying realizations of the processing time. In this way both solutions will be tested in an equal way, resulting in acceptance decisions that are more reliable.

An extra robustness tweak, to overcome strange realizations, is the application of a cutoff strategy. When the cutoff strategy is applied, only a fraction of all samples are regarded. The cutoff can for example disregard the top 20% worst samples (largest processing times) and the bottom 30% of the best samples. This technique effectively samples results in the region where robustness can be found at its best. Section 2.3.2 gives an overview of the details for this heuristic.

The results of combining the *Dominance Test* with the *Cutoff Sampling Heuristic* can be found in Table 2.5. The columns *LS10 /w dom.test* and *LS5 /w dom.test* give an overview of the results obtained by applying the *Dominance Test* and *Cutoff Sampling* heuristics. We can clearly see that the results for this combination of heuristics performs better than the classical heuristics. Overall we can say that when applying this combination of heuristics the local search will be more robust and has a better convergence character.

2.5.2 Quantitative Results of the Selected Methods

Since the methods from the previous section are only applied to a single instance, we still need to check what the performance will be for different situations. In this section we will apply a selection of the methods to instances that will be varied on their underlying disturbances and selections of disturbance sets. It is clear that the behaviour of the local search highly depends on how the underlying instances behave, therefore this experiment will give us a greater insight of the performance of the methods.

The instances selected for this quantitative experiment are listed in Table 2.1. The selected methods for this quantitative experiment consists of the following list:

- Percentiles 60, 70, 80, 90
- Factors 1.1, 1.2, 1.3
- 100 processing times sampling

- LS with 1 time sampling
- LS with 5 times sampling, dominance test and cutoff sampling
- LS with 10 times sampling, dominance test and cutoff sampling

Table 2.6 gives an overview of the results for the job disturbance instances. Table 2.7 gives an overview of the results for the machine disturbance instances. Table 2.8 gives an overview of the results for the total disturbance instances.

2.5.3 Results of the Experiments on Scaled Instances

The previous results clearly give an indication on the best performing methods. However in the previous series of experiments, the problem instances were very moderate in terms of job shop size and processing time span. The final step in determining the performance of the optimization methods is to apply the methods to instances for which some properties are scaled up, in order to create larger and more difficult instances. We already know that the best performing methods can be applied to a variety of probability distributions, therefore we focus on increasing the amount of jobs and machines in the Job Shop instances. Also the range of base processing times, the processing times that define the mean of the probability distributions, are scaled. Table 2.9 describes the details of the instances that are selected for this experiment. Note that each instance is generated randomly, therefore the schedules have no corresponding properties.

Increased Job Shop Size Instances

The first set of scaled instances is based on larger instances. Until now only Job Shops consisting of 10 machines and 10 jobs have been considered. By increasing both the amount of jobs and machines, the problem becomes larger and therefore more difficult to solve.

As can been seen from Table 2.9, all operations in the set *Large* receive an Exponential processing time with averages ranging from 10 to 30 time units. Since the problems become harder to solve, the local search takes more time to reach an optimum. To keep the experiment practicable, we have limited the amount of iterations to a maximum of 20000, also only 5 restarts are executed.

For each instance we applied the local search with *Five Times Sampling*, *Ten Times Sampling* and with a *Percentile Robustness* of percentile 70. Each setting is executed five times, resulting in five possibly different schedules. Finally, each schedule is simulated 1000 times, from which an average makespan is obtained. Table 2.10 shows the quantitative results of this experiment.

Increased Span of Processing Times

The second set of instances are based on large processing time averages. For the Exponential distribution, due to the large variance, samples of the processing time distributions diverge to a great extent. To force the instances to behave more natural, we have chosen to apply the 4-Erlang distribution. The parameters have been estimated such that the mean of the distribution equals the base processing time. For the 4-Erlang distribution this means that the variance is four times smaller compared to the Exponential distribution with the same mean.

By applying the best methods to instances with large processing times we can check whether the behaviour of the methods is influenced. Table 2.11 shows the results on the experiments with increased span on the processing times.

2.5.4 Analysis

In the previous subsections several methods have been developed to support the optimization of stochastic Job Shop instances. By running experimental analysis on the results we were able to determine the relative performance of the schedules. In this final subsection we will discuss the results of the previous experiments.

Determination of Methods

The first phase explored the possibilities of several methods. In this phase the classical methods were compared to more sophisticated methods. It is clear from the beginning of the first phase that applying a sampling technique, without further supporting it with other local search components, is not able to produce robust makespan averages. Also the method of applying sampling methods to individual processing time distributions does not produce the desired results.

An important observation of the first phase was the fact that the local search application with the *Single Sampling* method (*LS1*), observed very promising global optima. During the local search the optimum values of the makespans dominated the values of the classical methods. This observation actually gives a clear indication why a single sample based result is biased. The *Single Sampling* method at some points observes rather optimistic processing times. As a result, the local search will adopt the current solution as global best solution. If in later iterations of the local search the simulation will observe 'normal' realizations, the schedules will most probably be outperformed by the current global best, which is actually a too optimistic evaluation of a schedule.

We can generalise this idea even further. Since local search methods are designed to accept solutions that are better, any evaluation of the schedule will be accepted as long as its score is better than the current global best. Even if the evaluation is biased, the local search will still accept the solution. In later iterations the local search is unable to withdraw the biased results, since all other evaluations our being outperformed.

In a first attempt to overcome this problem, we applied *Multiple Times Sampling*. From the results in Table 2.3 we can see that the more we applied sampling, the better decisions the local search algorithm has made. Still the *Ten Times Sampling* method is not able to produce better schedules.

When multiple samples are considered by the local search, an additional decision needs to be taken. For a local search algorithm, the way to obtain new solutions is to move to a neighbour solution. A neighbour solution is obtained by altering the current solution in specific ways. For each of the sample results, neighbour moves can be obtained. Table 2.4 shows the results of applying the *Multiple Times Sampling* method in combination with a *Neighbourhood Repository*. The *Neighbourhood Repository* contains neighbour moves obtained from all the samples. From the results in the table we can conclude that there is no significant difference when the *Neighbourhood Repository* is applied.

In the final set of experiments from the first phase, a *Dominance Test* is applied. The *Dominance Test* gives the local search a better indication on the relative performance of the schedules. When two solution are compared, the test simulates both schedules based on common random numbers. In some way this technique is also based on the application of samples, however important for the *Dominance Test* is the fact that the schedules will retrieve common random number realizations. Effectively this temporarily considers the schedules as being deterministic instances

since both schedules receive the same deterministic processing time value for operations. If in the process of dominance testing optimistic processing time realization are observed, the test can still determine which of the two schedules performs better. This also includes realization that can be considered too pessimistic. From Table 2.5 we can conclude that the *Dominance Test* is able to produce robust makespan averages that score better than the classical methods.

Quantitative Results of the Selected Methods

Phase two describes the quantitative results of the selected methods. Interesting to see is that for most of the simulation results, the difference in performance between the *LS5* and *LS10* methods is relatively small. Also in a several results the performance of the *LS5* method is better than the *LS10* method. This suggests that taking only five samples is sometimes sufficient for the search process.

Remarkable is the fact that the LogNormal distribution for disturbance set C in Table 2.6 has problems finding good solutions for the sampling methods. For this specific instance the *Factor 1.3* method performs best. A similar behaviour can be found in for the results of the machine disturbances in Table 2.7, where for instance *mdLogNormalC Factor 1.1* performs best. Comparing the results in Table 2.6 per distribution type, we see that for the Erlang distribution the difference between the *LS5* and *LS10* methods compared to the other methods is very large. The relative performance between the methods for the Uniform distribution are the most moderate, compared to the other distributions.

For the total disturbance instances in Table 2.8, the distinction between the methods is clear. In this table the *Five Times Sampling* (LS5) and *Ten Times Sampling* (LS10) methods have a much better performance compared to the other methods. This indicates that the more stochastic processing times are involved, the better the sampling methods work.

From this set of results it is clear that a local search process, supported by the sampling methods and dominance test, will give better results compared to the classical approaches.

Increased Job Shop Size Instances

For the larger instances it is interesting to see whether the sampling methods keep performing well. From the increased Job Shop size results in table 2.10 we first focus on the relative performance of the worst schedule found by the sophisticated methods and the best schedule found by the classical method.

Instance *tdLarge15x10* still shows results that overlap. The worst result found by *LS10*, $821,6$ has a large overlap with the best result found by *Percentile 70*. However if we keep increasing the instance size, these differences will diverge.

Already from instance *tdLarge15x15* the overlap is gone. Table 2.12 on page 47 gives an overview of the gaps between the *LS5* and *Percentile 70* methods. If the gap is negative there is an overlap, otherwise there is no overlap, indicating that all average results of one method are strictly smaller than the results of the other method.

Also the performance between the worst result of *LS10* and the best results of *LS5* tends to diverge, however the overlap of performance still remains. This again gives the indication that five times sampling performance ratio is worth mentioning and keeps performing for most instances.

Finally the span between the average base processing times are analyzed. Table 2.11 gives similar results compared to the large instance experiments. The differences between the two sampling methods stays roughly the same for the various instances.

2.5.5 Relative Performance of the Classical Cool Down Approach

Each of the previously described results were obtained by applying the Simmulated Annealing local search heuristic. The approach taken for accepting worse solutions during the search process differs from the classical approach. In order to identify the performance difference for this approach, several experiments with the classical approach are executed. Table 2.13 describes the results for these experiments.

For each combination the local search algorithm is applied five times. Next the five result schedules are simulated 1000 times, from which an average makespan is obtained. The results in the table are based on the average makespan results.

When looking at the results in Table 2.13 we see that when the temperature is chosen correctly, the average makespan is better for the classical method. Although the minimum found by our non-standard approach is relative close to the best performing standard method, we can not draw any conclusion on the results without a proper statistics analysis.

Our approach is not optimal for the outcome of the search process, however this approach does allow us to compare the results between several local search processes, without being influenced by the stochastic realizations.

2.6 Conclusion and Further Research

In the experimental analysis of the Stochastic Job Shop Scheduling different methods have been employed. The development of these methods were aimed at supporting the local search process in finding good solutions for the stochastic instances. By applying different variants of the methods and running extensive experiments, the best combinations of methods is found.

Overall we can conclude that the methods *Five Times Sampling* and *Ten Times Sampling* are to be considered the best performing methods. Note that these methods can only be applied under the support of a *Dominance Test*. Furthermore, the *Cutoff Sampling* can be used in order to bias the probability distributions. By tweaking the *Cutoff Sampling* specific regions of the probability distributions are tried to be explored. A negative side effect of the additional *Cutoff Sampling* is the fact that only a small subset of the results are considered. Extra computation time, used for calculating the results that are not considered, is wasted.

2.6.1 Estimation of Percentiles

In order to see why the result sampling technique is profitable, we look at how we can solve the Stochastic Job Shop Problem by estimating the percentiles of the underlying processing time variable. Take for example the Stochastic Job Shop instance from Section 2.5, used for determining the combination of solution approaches. In order to solve this problem by using deterministic processing times, we first need to estimate a percentile that produces good results when simulating the schedule 1000 times. This approach requires the local search to be executed numerous times with fixed percentiles. A binary search between a minimum and maximum percentile can be performed, however the landscape of average makespans does not require to

have a unique global minimum. The ranges of percentiles therefore has to be explored to a great extent.

Additional to the estimation of the percentiles we can execute a local search process that uses the results sampling technique. This approach requires only a limited amount of search processes in order to determine the final schedule. Figure 2.4 givens an overview of the percentile estimation for Percentile Robustness and the Ten Times Sampling technique with dominance test.

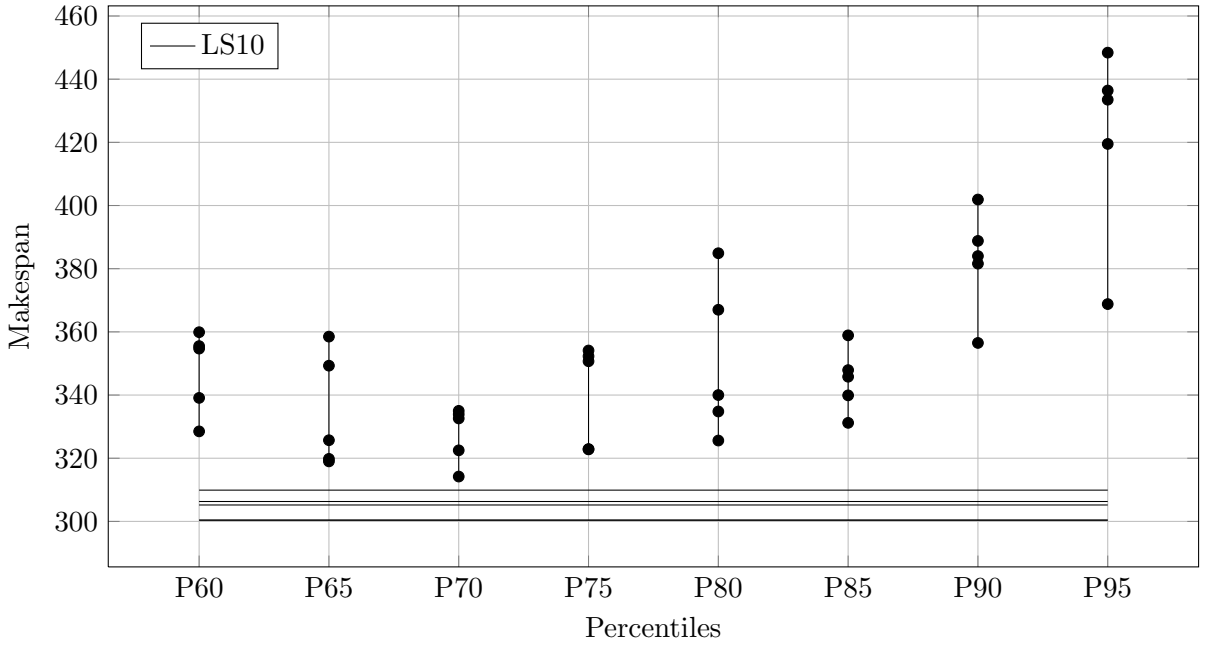


Figure 2.4: Overview of the Percentile Robustness method compared to the Ten Times Sampling method. The data represented in this figure is based on the mean makespan of 1000 simulations. Each method is executed five times, resulting in five schedules.

The figure shows the results of 40 Percentile Robustness experiments, indicated by the connected dots, and 5 results for the Ten Times Sampling experiments, indicated by the horizontal lines. As we can see, the Ten Times Sampling technique produces the best results compared to the results of the estimated percentiles. Also the span between the minimum and maximum average for the Ten Time Sampling approach is smaller than any of the spans for the minimum and maximum average results for the percentiles. By applying the sampling techniques from this research, we can effectively optimize stochastic instances with a limited amount of computing power. The solutions that are found by this technique outperform the classical techniques in both solution optimality and required computing time.

2.6.2 Lucky Ticket Theory

From the research we employed in this thesis, we can conclude that the application of Multiple Times Sampling in only possible in combination with a Dominance Test. This is especially true since we use samples in a local search algorithm. In order to see how this concept relates to the Lucky Ticket Theory, Figure 2.5 first presents the probability density function of the different probability distributions. Again, the mean of the various probability distributions is in all cases equal to 15.0.

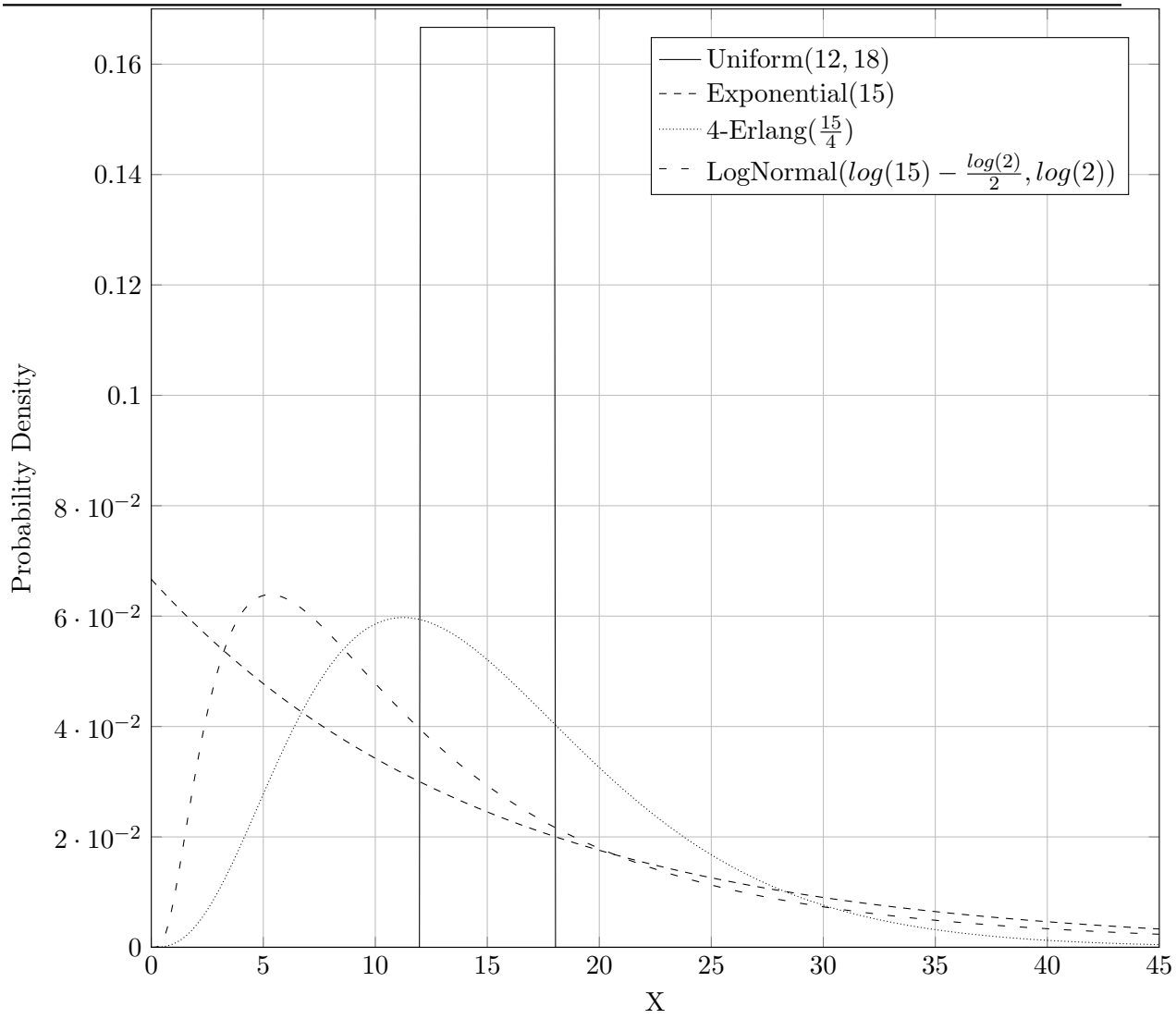


Figure 2.5: Probability density of the selected probability distributions. For each distributions the mean equals 15.0

Consider the Exponential(15) distribution. For this distribution the smallest processing times receive a large probability, relative to the probability of large processing times. When determining various samples of this distribution, we already know that the average of 15 is found at percentile $63,2$. Therefore, a large portion of the samples will be optimistic.

If for a Job Shop instance the processing times have smaller realizations, this will not influence the makespan in a negative way, since the smaller processing times can only decrease the makespan objective value. However, we try to optimize the schedule for processing times that are larger than the distribution's mean. The performance of the final objective value, based on 1000 simulations, will highly depend on the realizations that are larger than the distribution's mean. Therefore we would like the local search process to observe processing times realizations that are larger than the distribution's mean in order to optimize for average and worser case solutions.

In applying samples in the local search we find that the optimistic realizations will force the search process to adopt these smaller realizations. This is caused by the greedy behaviour of the algorithms since better solutions are always accepted. We call the optimistic realizations

"Lucky Tickets". The problems that arise when lucky tickets are adopted as local or global best is that these solutions will never be disregarded, even though the solution can be considered lucky. Also for other distributions in Figure 2.5, the 50% percentile is smaller than the mean.

The opposite effect of the lucky ticket theory is similar. If a schedule is given bad realizations, the schedule will never be accepted. The Dominance Test proves to be very effective against the wrong decisions. When comparing candidate solutions on basis of common random numbers, the relative performance of lucky tickets or bad tickets is used. Therefore when determining which of the two solutions is better, it does not depend on the actual realization but on how robust the schedules are for these realizations.

Finally, for the Job Shop Scheduling problem the neighbour solutions are derived from the current result. Lucky Tickets can also 'poison' the available neighbours, therefore when sampling is applied, neighbours applicable in all sample results must be included in the search process.

2.6.3 Further Research

The application of *Multiple Times Sampling* requires a significant amount of computing power. Also when the *Cutoff Sampling* is employed, some of the computation time is wasted. In order to make the sampling techniques applicable for large instances, the computation time must be optimized.

The important concept in this combination of *Stochastic Local Search* is that the decisions and the objective function are based on multiple realizations of the underlying probability distribution. For example we could for each processing time distribution determine a number of realizations. These realizations can be stored and used when different samples need to be determined.

First of all note that if such an approach is used, we lose the ability to observe large variations. To overcome this limitation, large amount of samples can be calculated, from which we then apply a cutoff. The fact that the realizations of the probability distributions are predetermined gives us the advantage that the computation time to determine these numbers is constant during the search process. In order to produce good schedules, the *Dominance Test* must still be used based on common random realizations. In detail this means that the schedules that need to be compared must use the same processing time realizations from the pool of realizations.

The variance and unforeseen characteristics of the stochastic instances can be mimicked by making sure to take random samples from the pool of realizations. This will also influence the mutual effects that play an important role in Job Shop Scheduling. The question is however if the random samples from the predetermined pool are representative enough since we can only have a limited amount of realizations.

Name	Size	Instance	Disturbance type	Disturbance set
jdTest	10x10	jdTest	Exponential	jobs 4, 6, 9
jdUniformA	10x10	jdA	Uniform	jobs 1, 5, 9
jdUniformB	10x10	jdB	Uniform	jobs 0, 2, 4, 6, 8
jdUniformC	10x10	jdC	Uniform	jobs 0, 1, 3, 4, 6, 7, 9
mdUniformA	10x10	mdA	Uniform	machines 1, 5, 9
mdUniformB	10x10	mdB	Uniform	machines 0, 2, 4, 6, 8
mdUniformC	10x10	mdC	Uniform	machines 0, 1, 3, 4, 6, 7, 9
tdUniformX	10x10	tdX	Uniform	all operations
jd4ErlangA	10x10	jdA	4-Erlang	jobs 1, 5, 9
jd4ErlangB	10x10	jdB	4-Erlang	jobs 0, 2, 4, 6, 8
jd4ErlangC	10x10	jdC	4-Erlang	jobs 0, 1, 3, 4, 6, 7, 9
md4ErlangA	10x10	mdA	4-Erlang	machines 1, 5, 9
md4ErlangB	10x10	mdB	4-Erlang	machines 0, 2, 4, 6, 8
md4ErlangC	10x10	mdC	4-Erlang	machines 0, 1, 3, 4, 6, 7, 9
td4ErlangX	10x10	tdX	4-Erlang	all operations
jdLogNormalA	10x10	jdA	LogNormal	jobs 1, 5, 9
jdLogNormalB	10x10	jdB	LogNormal	jobs 0, 2, 4, 6, 8
jdLogNormalC	10x10	jdC	LogNormal	jobs 0, 1, 3, 4, 6, 7, 9
mdLogNormalA	10x10	mdA	LogNormal	machines 1, 5, 9
mdLogNormalB	10x10	mdB	LogNormal	machines 0, 2, 4, 6, 8
mdLogNormalC	10x10	mdC	LogNormal	machines 0, 1, 3, 4, 6, 7, 9
tdLogNormalX	10x10	tdX	LogNormal	all operations
jdExponentialA	10x10	jdA	Exponential	jobs 1, 5, 9
jdExponentialB	10x10	jdB	Exponential	jobs 0, 2, 4, 6, 8
jdExponentialC	10x10	jdC	Exponential	jobs 0, 1, 3, 4, 6, 7, 9
mdExponentialA	10x10	mdA	Exponential	machines 1, 5, 9
mdExponentialB	10x10	mdB	Exponential	machines 0, 2, 4, 6, 8
mdExponentialC	10x10	mdC	Exponential	machines 0, 1, 3, 4, 6, 7, 9
tdExponentialX	10x10	tdX	Exponential	all operations

Table 2.1: Stochastic instances for which the experiments are conducted. The *Name* column describes the unique identifier for an experiment, the *Instance* column describes the underlying generated instance. The seven deterministic instances (jdA, jdB, jdC, mdA, mdB, mdC, tdX) are used for each different probability distribution.

Percentile 70		Factor 1.2		Percentile 60		Percentile 80		Percentile 90		LS1	
Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
317,4	37,6	330,1	36,5	328,0	40,1	326,4	33,5	356,7	26,4	369,9	47,4
321,8	36,6	335,1	37,7	337,0	41,8	334,7	31,6	381,9	30,6	400,8	42,8
331,1	37,4	337,6	36,8	355,1	46,9	340,1	35,7	382,5	31,6	388,9	44,3
333,1	37,8	348,2	37,6	355,8	45,1	366,1	30,5	389,8	31,0	368,8	38,5
336,5	36,7	349,4	38,5	360,9	42,1	384,3	31,0	400,8	23,4	397,6	54,0
328,0		340,1		347,4		350,3		382,3		385,2	

Table 2.2: Initial research results after applying the local search methods *Single Sampling (LS1)*, *Factor* and *Percentile*. Each individual result consists of the average over 1000 simulation runs on the same schedule. The individual results are sorted from small to large average makespan. The bottom line shows the average over the five experiments.

Percentile 70		100proc		LS10		LS5		LS1	
Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev	Average	Std. Dev	Average	Std. Dev.
317,4	37,6	328,7	41,3	351,6	39,1	357,0	40,0	368,8	38,5
321,8	36,6	339,1	42,5	353,5	40,2	358,2	41,8	369,9	47,4
331,1	37,4	352,3	46,4	361,4	42,6	359,7	48,9	388,9	44,3
333,1	37,8	356,0	41,4	363,0	37,5	382,0	48,0	397,6	54,0
336,5	36,7	363,2	44,1	365,1	40,1	390,1	41,1	400,8	42,8
328,0		347,9		358,9		369,4		385,2	

Table 2.3: Research results after applying the local search methods *Single Sampling*, *Factor*, *Five Times Sampling*, *Ten Times Sampling* and *100 Processing Times Sampling*.

Percentile 70		100proc		LS10		LS5 /w rep.		LS5		LS10 /w rep.	
Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev	Average	Std. Dev.	Average	Std. Dev	Average	Std. Dev.
317,4	37,6	328,7	41,3	351,6	39,1	350,7	40,8	357,0	40,0	362,3	39,7
321,8	36,6	339,1	42,5	353,5	40,2	361,2	40,7	358,2	41,8	365,7	36,7
331,1	37,4	352,3	46,4	361,4	42,6	364,8	46,3	359,7	48,9	366,5	38,6
333,1	37,8	356,0	41,4	363,0	37,5	375,9	45,7	382,0	48,0	368,3	44,2
336,5	36,7	363,2	44,1	365,1	40,1	389,9	44,0	390,1	41,1	388,6	46,2
328,0		347,9		358,9		368,5		369,4		370,3	

Table 2.4: Research results after applying the local search heuristic *Multiple Times Sampling* with the additional heuristic of employing a neighbourhood repository filled by neighbours based on all samples. The *Multiple Times Sampling*, the *Percentile 70* and the *100 Processing Times Sampling* results from the previous experiments are also included.

LS10 /w dom.test		LS5 /w dom.test		Percentile 70		100proc		LS5 /w rep.		LS10 /w rep.	
Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
300,4	37,4	300,0	42,8	317,4	37,6	328,7	41,3	350,7	40,8	362,3	39,7
300,4	38,9	300,5	40,1	321,8	36,6	339,1	42,5	361,2	40,7	365,7	36,7
305,2	40,6	306,9	44,0	331,1	37,4	352,3	46,4	364,8	46,3	366,5	38,6
306,3	42,1	312,7	38,6	333,1	37,8	356	41,4	375,9	45,7	368,3	44,2
309,9	42,5	329,5	43,9	336,5	36,7	363,2	44,1	389,9	44	388,6	46,2
304,4		309,9		328		347,9		368,5		370,3	

Table 2.5: Research results after applying the local search heuristic *Multiple Times Sampling* with the additional heuristic of applying a dominance test and cutoff sampling. Previous results determinded by the *Percentile 70*, *100 Processing Times Sampling* and the *Multiple Times Sampling* with neighbourhood repository are also included.

Instance	100proc	LS1	LS5	LS10	P60	P70	P80	P90	Factor1.1	Factor1.2	Factor1.3
jd4ErlangA	233,2	242,8	219,7	227,6	238,8	260,1	260,3	269,7	252,2	258,1	262,1
	106,1%	110,5%	**100,0%**	103,6%	108,7%	118,4%	118,5%	122,8%	114,8%	117,5%	119,3%
jd4ErlangB	220,6	225,4	192,9	194,4	234,2	229,5	256,0	260,7	223,7	233,0	252,3
	114,4%	116,9%	**100,0%**	100,8%	121,4%	119,0%	132,7%	135,2%	116,0%	120,8%	130,8%
jd4ErlangC	169,6	173,5	163,0	162,7	198,8	209,0	215,7	240,8	200,4	223,6	221,1
	104,2%	106,6%	100,2%	**100,0%**	122,2%	128,4%	132,5%	147,9%	123,1%	137,4%	135,9%
jdExponentialA	332,9	387,4	307,3	303,3	332,0	313,2	313,7	358,8	322,3	319,8	323,8
	109,8%	127,7%	101,3%	**100,0%**	109,5%	103,3%	103,4%	118,3%	106,3%	105,4%	106,8%
jdExponentialB	416,5	585,1	350,6	342,2	370,9	360,0	373,6	377,3	356,3	356,0	361,6
	121,7%	171,0%	102,5%	**100,0%**	108,4%	105,2%	109,2%	110,2%	104,1%	104,0%	105,7%
jdExponentialC	382,1	517,3	357,3	352,3	380,1	371,0	386,7	377,3	353,8	371,5	361,8
	108,4%	146,8%	101,4%	**100,0%**	107,9%	105,3%	109,8%	107,1%	100,4%	105,4%	102,7%
jdLogNormalA	348,4	397,0	305,6	300,5	332,2	342,8	309,9	333,3	337,9	315,9	316,3
	115,9%	132,1%	101,7%	**100,0%**	110,5%	114,1%	103,1%	110,9%	112,4%	105,1%	105,2%
jdLogNormalB	384,6	520,8	347,0	340,4	356,7	348,4	361,5	378,4	347,0	350,5	358,4
	113,0%	153,0%	101,9%	**100,0%**	104,8%	102,3%	106,2%	111,2%	101,9%	103,0%	105,3%
jdLogNormalC	376,8	497,3	354,3	355,0	352,9	372,1	364,3	374,2	355,5	366,4	350,6
	107,5%	141,8%	101,1%	101,2%	100,7%	106,1%	103,9%	106,7%	101,4%	104,5%	**100,0%**
jdUniformA	259,5	279,5	242,0	253,4	263,1	260,2	258,0	275,7	262,2	266,3	268,5
	107,2%	115,5%	**100,0%**	104,7%	108,7%	107,5%	106,6%	113,9%	108,4%	110,1%	111,0%
jdUniformB	293,4	326,1	239,2	242,9	267,3	257,0	263,0	263,7	282,1	285,7	276,3
	122,7%	136,3%	**100,0%**	101,6%	111,8%	107,5%	110,0%	110,3%	118,0%	119,4%	115,5%
jdUniformC	238,9	288,6	236,0	237,0	251,5	266,2	273,1	253,4	257,0	278,1	269,6
	101,3%	122,3%	**100,0%**	100,5%	106,6%	112,8%	115,7%	107,4%	108,9%	117,9%	114,3%

Table 2.6: Research results after applying the selected heuristics for the job disturbances sets. For each instance the relative performance is displayed in the second row.

Instance	100proc	LS1	LS5	LS10	P60	P70	P80	P90	Factor1.1	Factor1.2	Factor1.3
md4ErlangA	242,2	259,6	196,5	204,0	244,0	246,4	300,0	277,4	256,3	239,4	269,1
	123,3%	132,1%	**100,0%**	103,8%	124,2%	125,4%	152,7%	141,1%	130,4%	121,8%	136,9%
md4ErlangB	229,8	234,3	211,7	212,2	247,1	260,2	249,6	282,6	249,6	259,0	258,2
	108,6%	110,7%	**100,0%**	100,2%	116,7%	122,9%	117,9%	133,5%	117,9%	122,4%	122,0%
md4ErlangC	162,3	191,9	151,3	151,4	188,7	181,0	183,8	192,8	186,8	188,1	187,3
	107,3%	126,9%	**100,0%**	100,1%	124,7%	119,7%	121,5%	127,5%	123,5%	124,3%	123,8%
mdExponentialA	361,9	455,9	289,0	285,2	315,5	312,8	396,0	430,6	306,9	305,9	348,4
	126,9%	159,9%	101,3%	**100,0%**	110,6%	109,7%	138,9%	151,0%	107,6%	107,3%	122,2%
mdExponentialB	401,5	472,5	349,4	347,9	385,2	384,1	387,8	424,6	377,1	375,6	381,8
	115,4%	135,8%	100,4%	**100,0%**	110,7%	110,4%	111,5%	122,0%	108,4%	107,9%	109,7%
mdExponentialC	391,0	534,5	361,8	359,6	360,3	359,2	373,1	387,7	375,5	364,4	362,5
	108,8%	148,8%	100,7%	100,1%	100,3%	**100,0%**	103,9%	107,9%	104,5%	101,4%	100,9%
mdLogNormalA	358,2	429,8	286,2	280,7	316,2	346,0	317,0	385,1	333,5	318,1	335,0
	127,6%	153,1%	102,0%	**100,0%**	112,7%	123,3%	112,9%	137,2%	118,8%	113,3%	119,4%
mdLogNormalB	401,8	455,0	355,2	343,3	373,9	379,8	382,0	394,9	356,3	379,9	372,8
	117,0%	132,5%	103,5%	**100,0%**	108,9%	110,6%	111,3%	115,0%	103,8%	110,6%	108,6%
mdLogNormalC	383,9	465,0	359,1	353,7	360,4	358,4	363,7	371,4	353,3	379,1	364,3
	108,7%	131,6%	101,7%	100,1%	102,0%	101,4%	102,9%	105,1%	**100,0%**	107,3%	103,1%
mdUniformA	259,1	262,9	233,4	228,7	259,1	262,1	271,1	247,9	265,8	268,9	281,6
	113,3%	114,9%	102,0%	**100,0%**	113,3%	114,6%	118,5%	108,4%	116,2%	117,6%	123,1%
mdUniformB	272,7	310,0	260,8	262,1	270,4	281,9	281,9	274,7	289,9	293,1	295,5
	104,6%	118,9%	**100,0%**	100,5%	103,7%	108,1%	108,1%	105,3%	111,2%	112,4%	113,3%
mdUniformC	248,5	274,2	235,1	234,7	247,8	261,5	257,5	255,3	256,2	257,1	254,7
	105,9%	116,8%	100,2%	**100,0%**	105,6%	111,4%	109,7%	108,8%	109,2%	109,5%	108,5%

Table 2.7: Research results after applying the selected heuristics for the machine disturbances sets. For each instance the relative performance is displayed in the second row.

Instance	100proc	LS1	LS5	LS10	P60	P70	P80	P90	Factor1.1	Factor1.2	Factor1.3
td4ErlangX	93,5	108,2	76,8	75,6	86,3	83,9	91,3	84,0	88,6	93,2	92,0
	123,7%	143,3%	101,6%	**100,0%**	114,2%	111,0%	120,8%	111,2%	117,3%	123,4%	121,8%
tdExponentialX	479,6	575,5	403,7	400,5	437,3	444,5	455,2	444,5	451,7	465,6	451,6
	119,7%	143,7%	100,8%	**100,0%**	109,2%	111,0%	113,7%	111,0%	112,8%	116,3%	112,8%
tdLogNormalX	472,9	541,8	413,1	399,4	440,7	440,4	432,3	446,6	442,4	451,5	447,8
	118,4%	135,7%	103,4%	**100,0%**	110,3%	110,3%	108,2%	111,8%	110,8%	113,0%	112,1%
tdUniformX	301,1	328,0	252,1	255,7	312,1	296,6	316,1	306,5	306,8	311,9	314,7
	119,4%	130,1%	**100,0%**	101,4%	123,8%	117,6%	125,4%	121,6%	121,7%	123,7%	124,8%

Table 2.8: Research results after applying the selected heuristics for the total disturbances sets. For each instance the relative performance is displayed in the second row.

Name	Size	Disturbance type	Mean Processing Times	Disturbance set
tdLarge15x10	15x10	Exponential	10 – 30	all operations
tdLarge15x15	15x15	Exponential	10 – 30	all operations
tdLarge20x15	20x15	Exponential	10 – 30	all operations
tdLarge20x20	20x20	Exponential	10 – 30	all operations
tdSpan10-40	10x10	4-Erlang	10 – 40	all operations
tdSpan10-80	10x10	4-Erlang	10 – 80	all operations
tdSpan10-120	10x10	4-Erlang	10 – 120	all operations

Table 2.9: Various stochastic instances with an increased amount of jobs and machines or processing time averages. For each instance all the operations receive a stochastic processing time defined by Exponential distribution for the larger instances and the 4-Erlang distribution for the instances with a larger processing time span. The *Large* instances have an increased Job Shop size. The *Span* instances have increased average processing times.

Instance	LS10 Average	LS10 Std. Dev.	LS5 Average	LS5 Std. Dev.	Percentile 70 Average	Percentile 70 Std. Dev.
tdLarge15x10	781,7	93,0	804,7	97,7	792,6	95,5
	804,8	103,4	807,0	101,8	803,5	100,4
	807,2	103,4	815,9	101,5	810,7	100,0
	807,5	101,4	824,6	104,1	837,5	102,0
	821,6	99,3	836,1	104,8	874,6	106,4
	804,6		817,7		823,8	
tdLarge15x15	932,8	101,9	970,0	108,9	1091,6	107,6
	941,7	100,8	979,6	104,0	1103,7	114,7
	956,3	102,3	994,2	103,8	1105,5	116,5
	964,4	105,1	999,3	107,4	1128,6	114,5
	986,4	107,9	1003,2	111,8	1179,7	122,0
	956,3		989,3		1121,8	
tdLarge20x15	931,7	94,6	943,4	96,5	1297,2	119,5
	936,0	93,2	955,0	98,0	1307,7	119,1
	938,7	97,7	955,8	94,8	1318,9	120,0
	948,5	96,4	971,2	95,3	1358,4	123,7
	949,2	96,2	997,2	100,5	1408,5	129,2
	940,8		964,5		1338,1	
tdLarge20x20	1176,7	104,7	1199,4	103,0	1791,9	142,5
	1177,9	108,1	1208,7	107,6	1821,7	146,5
	1190,9	109,1	1220,0	105,6	1829,9	148,1
	1200,0	106,6	1236,5	109,6	1840,0	150,5
	1200,3	109,6	1256,0	114,7	1893,2	148,3
	1189,2		1224,1		1835,3	

Table 2.10: Research results after applying the local search methods *Ten Times Sampling*, *Five Times Sampling* and *Percentile Robustness* with percentile 70. Each individual result consists of the average over 1000 simulation runs on the same schedule. The individual results are sorted from small to large average processing times. The bottom line shows the average over the five experiments.

Instance	LS10 Average	LS10 Std. Dev.	LS5 Average	LS5 Std. Dev	Percentile 70 Average	Percentile 70 Std. Dev.
tdSpan10-40	111,9	9,4	111,8	9,3	114,3	9,6
	112,0	9,8	113,1	9,5	121,2	10,8
	112,3	10,0	114,3	9,7	125,1	10,5
	113,3	9,6	116,5	10,3	125,6	10,8
	114,0	9,6	118,2	10,3	139,3	11,2
	112,7		114,8		125,1	
tdSpan10-80	229,6	20,8	233,2	21,4	241,5	20,6
	230,7	21,6	233,5	22,6	242,3	21,0
	230,7	20,3	235,3	21,5	242,4	20,9
	231,9	20,5	239,7	21,0	259,6	22,7
	233,3	21,8	241,1	20,8	263,1	23,9
	231,2		236,6		249,8	
tdSpan10-120	319,0	28,5	319,2	30,4	355,8	30,4
	322,6	28,7	325,2	29,9	355,8	30,7
	323,2	29,4	330,1	29,6	383,8	31,2
	328,8	29,7	331,0	29,7	434,9	35,1
	334,7	29,9	339,9	29,5	443,5	36,5
	325,7		329,1		394,8	

Table 2.11: Research results after applying the local search methods *Ten Times Sampling*, *Five Times Sampling* and *Percentile Robustness* with percentile 70. Each individual result consists of the average over 1000 simulation runs on the same schedule. The individual results are sorted from small to large average processing times. The bottom line shows the average over the five experiments.

Instance	Worst Score LS5	Best Score Percentile 70	Gap
tdLarge15x10	836,1	792,6	-43,5
tdLarge15x15	1003,2	1091,6	88,4
tdLarge20x15	997,2	1297,2	700,0
tdLarge20x20	1256,0	1791,9	535,9

Table 2.12: Overview of the average makespan gaps between the worst solution found by LS5 and the best solution found by Percentile 70. A positive gap indicates no overlap between the results of the methods.

Method	Temperature	Avg	Min	Max	Std. Dev.
Probability Number	0,98	309,9	300,0	329,5	12,1
Classical	10	299,8	296,9	301,4	1,8
Classical	25	302,5	295,3	307,0	5,4
Classical	50	305,7	301,5	308,4	3,5
Classical	100	305,3	301,9	308,1	2,9
Classical	250	313,2	306,9	322,8	6,7
Classical	500	322,2	313,4	328,1	5,8

Table 2.13: Results after applying the local search algorithm with the five times sampling solution approach to the instance *jdTest* for several initial temperatures. The *Method* column indicates whether the classical method or the new method is used. The column *Temperature* describes the initial temperature for the search process.

Chapter 3

Blocking Job Shop Scheduling

"I don't think there's a punch-line scheduled, is there?"
— *Monty Python*

3.1 Introduction

In this chapter new approaches for solving the Blocking Job Shop are discussed. The Blocking Job Shop model is derived from the Standard Job Shop Scheduling Problem by adding an additional set of blocking constraints to the problem description.

The addition of the blocking constraints implies that there is no buffer space available for certain machines. During the period the operation is unable to move to the successor machine, the operation remains on the current machine. Therefore this model can be applied to production environments that are limited in the same way. A good example is the application of the Blocking Job Shop model for shunting movements of trains in railway stations [26]. Also production companies become more and more competitive, leading to limited or even the absence of buffers on the production line. For these problems the Blocking Job Shop becomes more applicable. As a consequence of adding extra constraints, the problem becomes harder to solve since more infeasibilities exist.

The literature proposes several methods to solve the Blocking Job Shop. Most of these methods consist of a branch and bound algorithm with branching on edges in the disjunctive graph. Also several local search algorithms have been proposed ([7], [16] and [8]). Corresponding to the disjunctive graph formulation, a Mixed Integer Program is formulated by Ramaswamy and Joshi [21] and further used by Van den Broek [26]. This formulation models both blocking and no-wait constraints.

Common in the characteristic available literature is that during the solution process, only feasible solutions are maintained. It is clear that when only considering feasible solutions, the amount of search freedom is limited. This leads to a complex search process and difficult neighbourhood operations.

To overcome these limitations and complexities we propose a new local search method. By temporarily allowing infeasible solutions, a broader landscape of solutions can be explored. In order to apply this new method, the MIP formulation from [26] is applied. Also a sophisticated method to calculate the objective function has been developed that at the same time records specific details of occurring infeasibilities.

An additional new local search method is the combination of the MIP formulation with

existing neighbourhood operations, to gain access to a larger neighbourhood. During the local search a neighbour solution can be obtained by for example inverting a disjunctive arc. Next the MIP formulation can be applied to optimize a small set of operations. The combination of local search neighbourhoods with the MIP formulation, improves the search process to find better solutions faster.

This chapter is structured as follows: first a formal definition of the Blocking Job Shop is given in Section 3.2, next Section 3.3 gives a cycle analysis, presenting an overview of several characteristics of infeasibilities specific to the Blocking Job Shop. Section 3.4 discusses the details of the solution approach. Next the results and analysis are discussed in Section 3.5, comparing the results with results from the literature. Finally Section 3.6 finishes this chapter a conclusion.

3.2 Formal Definition of the Blocking Job Shop

In this chapter we focus on solving the Blocking Job Shop Problem. This problem is derived from the Standard Job Shop Scheduling Problem, by adding an additional set of constraints. The Standard Job Shop Scheduling Problem uses the implicit assumption of unlimited intermediate buffers. When dropping this assumption, the storage location of jobs is becoming a part of the scheduling problem.

As a result of removing the unlimited buffers, machines become 'blocking' when unable to hand a finished job to its successor machine. Since no other storage locations other than the machines itself are available, the first action when completing an operation, is to check if the successor machine of the current job is available to process the job, and *block* until the successor machine becomes available otherwise.

When optimizing the Blocking Job Shop Problem we also have to take care of blocking times, which results in larger makespans for the schedules. Next to the blocking times, the additional set of constraints makes the problem harder to solve. The blocking constraints also introduce a new collection of infeasible solutions. Section 3.3 gives an overview of the various infeasibilities that can occur for the Blocking Job Shop.

Besides the infeasibilities of the Standard Job Shop Scheduling Problem, caused by a violation of job precedence constraints, the Blocking Job Shop also suffers from infeasibilities caused by blocking machines. The Blocking Job Shop Problem can be divided into two categories:

- Blocking Job Shop Problem *with swap*
- Blocking Job Shop Problem *without swap*

In the case with swap it is allowed to swap a set of jobs on machines that block on each other. This situation is characterized by the fact that for each machine the job currently on the machine is to be processed next on its successor machine. In the definition without swap, such actions are not allowed. A practical example is a manufacturing environment where the movements between the machines are handled by a single carrier such as a robot arm. In that case swaps are impossible. If every machine or stage would have its own carrier the swap can be executed since the jobs can be moved simultaneously. The two categories of Blocking Job Shop Problems are classified according to the following definition by Graham et al. [6]:

$$J \mid blocking \mid C_{max} \tag{3.1}$$

In this definition there is no distinction between the Blocking Job Shop Problem with swap or without swap. Additionally the *blocking* job characteristics can be extended with the *with swap* notation, to indicate a Blocking Job Shop where swaps are allowed.

Throughout this chapter we use an extended notation from Section 1.2.1. For a Blocking Job Shop with M machines and N jobs we have the following set of variables:

$O_{j,i}$ indicates operation number i of job j, where $i = 0 \dots M-1$ and $j = 0 \dots N-1$.

$p_{j,i}$ indicates the processing time of operation $O_{j,i}$.

$machine(o_{j,i})$ indicates the index of the machine that has to process operation $O_{j,i}$ where $machine(O_{j,i}) \in 0 \dots M-1$.

$s(O_{j,i})$ indicates the start time of operation $O_{j,i}$.

$e(O_{j,i})$ indicates the finishing time of operation $O_{j,i}$.

$w(O_{j,i})$ indicates the waiting time of machine $machine(O_{j,i})$, caused by waiting before the start of operation $O_{j,i}$ machine $machine(O_{j,i-1})$ first needs to finish operation $O_{j,i-1}$.

$b(O_{j,i})$ indicates the blocking time of machine $machine(O_{j,i})$, caused by blocking for machine $machine(O_{j,i+1})$ to take operation $O_{j,i+1}$.

See Figure 3.1 for an example of the above variables.

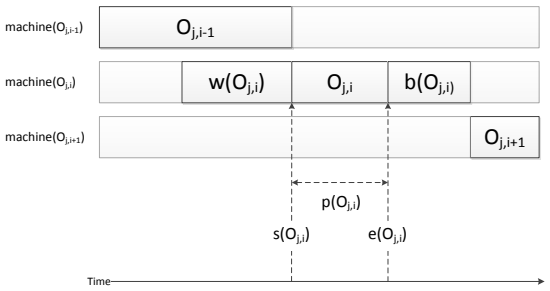


Figure 3.1: Example of the variable notation for the Blocking Job Shop Problem, used in this chapter.

From the above definition and the assumption that jobs visit machines only once, it is clear that we only consider pure rectangle instances. The amount of operations is a Blocking Job Shop is therefore defined by the following equation:

$$|O| = J \cdot M \tag{3.2}$$

Next to the waiting time of an operation, the Blocking Job Shop also introduces blocking time. For each machine, the amount of time the machine spends on processing operation $O_{j,i}$ is expressed by the following formula:

$$w(O_{j,i}) + p(O_{j,i}) + b(O_{j,i}) \tag{3.3}$$

The effective time that an operation $O_{j,i}$ resides on $machine(O_{j,i})$ is equal to $p(O_{j,i})+b(O_{j,i})$. Furthermore $e(O_{j,i}) = s(O_{j,i}) + p(O_{j,i})$, where the s and e functions respectively capture the start and finishing times of processing operation $O_{j,i}$.

The waiting time of operation $O_{j,i}$ is caused by the fact that its previous operation $O_{j,i-1}$ is not yet finished. If the waiting time equals zero, the machine immediately continues processing; in this case the previous operation has already finished and left the machine. If a job is unable to move directly to its the successor machine when finished, the job needs to stay on the machine. In the period the finished operation stays on the machine, the machine is 'blocked' to do effective work. The timespan between finishing the operation and removing the operation is therefore noted as blocking time $b(O_{j,i})$.

3.2.1 MIP Formulation

Also for the Blocking Job Shop Problems a MIP can be formulated. By extending the MIP formulation from Section 1.2.2, similar to [26], we are able to model both the two categories of Blocking Job Shop Problems. In the following definition set C consists of all conjunctive pairs in the disjunctive graph, set D consists of all the disjunctive pairs in the disjunctive graph and set O consists of all the operations defined in the job-shop:

$$\begin{aligned}
\min \quad & s_{n+1} && && (3.4)\\
\text{s.t.} \quad & \\
& s_{\sigma_i} \geq s_i + p_i && \forall (i,\sigma_i) \in C && (3.5)\\
& s_j \geq s_i + p_i - M(1 - x_{ij}) && \forall (i,j) \in D && (3.6)\\
& s_j \geq s_{\sigma_i} - M(1 - x_{ij}) + \epsilon && \forall (i,j) \in D && (3.7)\\
& x_{ij} + x_{ji} = 1 && \forall (i,j) \in D && (3.8)\\
& x_{ij} \in \{0,1\} && \forall (i,j) \in D && (3.9)\\
& s_j \in \mathbb{R}^+ && \forall j \in O && (3.10)
\end{aligned}$$

In the above definition, the variable set s_j models the start time of the operations. Variable σ_i indicates operation index for the successor operation of operation i. Furthermore, variables $x_{i,j}$ and $x_{j,i}$ model the disjunctive arcs for the two operations i and j, that are to be scheduled on a common machine:

$$x_{ij} = \begin{cases} 1 & \text{if operation } i \text{ is scheduled before operation } j \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

Constraint 3.5 ensures the precedence relations of the operations in a job. In this set of constraints also a constraint for the last operation of a job and the dummy operation $n+1$ is included. The dummy operation is equivalent to sink vertex t in the disjunctive graph. By minimizing the start time of the dummy operation s_{n+1} in Objective Function 3.4, we are effectively minimizing the makespan of the schedule.

Constraint 3.6 ensures that a machine can process only one operation at the time. Constraint 3.7 models the blocking constraints. If on a machine operation i is scheduled before operation j ($x_{ij} = 1$), operation i first has to be removed from the machine before being able to continue processing operation j. At some moment operation i is placed on the successor machine, thereby

starting the processing of the successor operation σ_i. The start time for the successor operation of i is indicated by s_{σ_i}. Since at s_{σ_i} the machine is empty, operation j can start. By constraining $s_j \geq s_{\sigma_i}$ we force the machine to be empty before the next operation in the schedule can be executed.

By giving M a big value, in our case the sum of all processing times $M = \sum_{o_j \in O} p_j$, the machine constraints (3.6) and the blocking constraints (3.7) can be automatically turned off in the case the constraints are not required. As stated in [26], Ramaswamy and Joshi [21] have proven that when error term ϵ is given a small value, an optimal deadlock-free schedule *without swap* can be found. The epsilon term turns the constraint in a strictly greater than constraint. As a side effect, the computation of the makespan (s_{n+1}) contains a small error. Between every pair of operations that is scheduled on the same machine the maximum error of 2ϵ can be found. To counteract this problem, ϵ can be chosen small enough so the error does not propagate to the objective function. Van Den Broek [26] states that choosing $\epsilon < \frac{1}{2n}$ counteracts this problem. However when applying this formulation during the optimization of various Blocking Job Shop Problem *without swap* instances, the solver sometimes ran into numerical problems.

The above formulation can be extended to conveniently calculate the machine schedules. When forming the machine schedules, the machine indexes of the operations determine the order of processing:

$$h_j = \sum_{(i,j) \in D} x_{ij} \qquad \forall j \in O, \forall (i,j) \in D \tag{3.12}$$

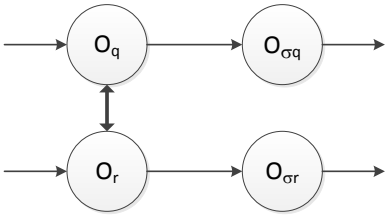
$$h_j \in \{0, \ldots, J-1\} \qquad \forall j \in O \tag{3.13}$$

Variable set 3.13 defines for each operation j a machine index h_j. If an operation receives a machine index of 0, that operation is scheduled first on $machine(o_j)$. A machine index of $J-1$ indicates that the operation is scheduled last on the machine, in this case the machine has $J-1$ predecessors scheduled before the operation. Constraint set 3.12 shows that counting how many predecessor operations are scheduled on the machine, determines the machine index.

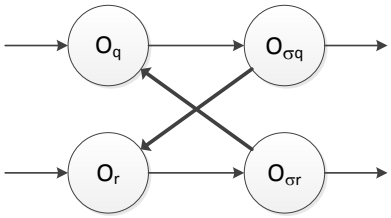
3.2.2 Alternative Graph

The disjunctive graph formulation, explained in Section 1.2.2, is unable to model the blocking constraints. In Section 3.3 a modification to the disjunctive graph is given, that allows infeasibilities regarding the blocking constraints to be found. From Chapter 1 we already know that for the Standard Job Shop Scheduling Problem there is a close relation between the disjunctive graph and the MIP formulation. In fact the MIP formulation for the Blocking Job Shop Problem can also be translated into a graph definition similar to the disjunctive graph.

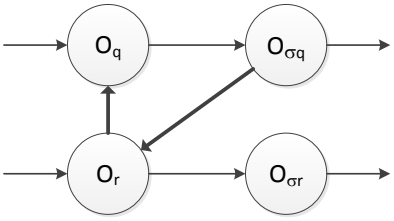
Mascis and Pacciarelli [16] defined the *alternative graph*, a generalization of the disjunctive graph, and gave a modification to the disjunctive arcs to model the blocking constraints. The blocking constraints in Equation 3.7 force operation j to start only if the successor operation of i, σ_i, is started on the successor machine. Instead of drawing the machine disjunction between operations on the machine, the disjunctions are now drawn between operations σ_i and j if $x_{ij} = 1$, or between σ_j and i if $x_{ji} = 1$. The final operation of a job is however non blocking, therefore the disjunctive arc that schedules the non blocking operation in front of another operation, receives the standard disjunction. The following list gives an overview of the case distinctions for the disjunctive arcs between operations q and r, that are to be scheduled on the same machine:



(a) Operations q and r are both non-blocking. The disjunctive arcs are similar to the disjunctive arcs from the disjunctive graph formulation.



(b) Both operations q and r are blocking, the disjunctions are drawn between the successor operation and the successor job.



(c) Only operation q is blocking. The disjunctions are defined by an arc between the successor operation of q (σ_q) and operation r for $x_{qr} = 1$, and between operation r and q for $x_{rq} = 1$.

Figure 3.2: Example of the three forms of disjunctive arcs for the alternative graph.

1. *q and r are both non-blocking.* The standard disjunctive arcs from the disjunctive graph are drawn: (q, r) and (r, q). See Figure 3.2a
2. *q and r are both blocking.* The disjunctions between the operation and the successor operation of the other operation are drawn: (σ_r, q) and (σ_q, r). See Figure 3.2b.
3. *q is blocking and r is non-blocking.* Only the disjunction from q to r is blocking: (σ_q, r) and (r, q). See Figure 3.2c.

The alternative graph formulation is however not used in this thesis. As we will see in Section 3.4.4, a custom implementation for the objective function is given that allows the access to detailed information about infeasible solutions and has a smaller runtime.

3.3 Cycle Analysis

For the Standard Job Shop Scheduling Problem, infeasible solutions are caused by violations of the job constraints. In order to detect these infeasibilities a longest path on the disjunctive graph can be calculated. If in this graph a cycle is found, the solution is infeasible. An alternative way to determine whether a solution to the Standard Job Shop Scheduling Problem is infeasible, is by calculating the makespan by means of a simulation. If the event list is empty while not all operations are processed, the solution can also be considered infeasible.

The introduction of the blocking constraints for the Blocking Job Shop Problem reduces the set of feasible solutions even more. In this section an analysis of the infeasible solutions for the Blocking Job Shop Problem is given. The analysis is based on previous research results [25]. By extending the disjunctive graph and formulating a dependency graph we are able to detect infeasibilities introduced by the blocking constraints.

3.3.1 Dependency Graph

In order to characterize infeasibilities, we need to structure the behaviour of a Job Shop Schedule. For the Standard Job Shop, infeasibilities occur by cycles in the disjunctive graph. By introducing a new graph formulation, that is also based on cycles, we are able to detect the additional set of infeasibilities introduced by the blocking constraints.

When utilizing a simulation model, users have detailed access to the intermediate states of the problem. By mimicking the behaviour of the simulation we are able to describe the additional set of infeasibilities. If the event list of a simulation is empty while there are operations that still need to be processed, a deadlock has occurred. For each machine we can identify the conditions that must be met in order to continue processing. We call these conditions *continuity dependencies*. By combining the continuity dependencies of all machines we form a *Dependency Graph*. Finally if a deadlock occurs in the Dependency Graph, the solution is infeasible. Analyzing the cycles on the Dependency Graph will identify the type of infeasibility.

Graph Definition

Given a Blocking Job Shop instance with m machines we define for each machine k a vertex V_k and a dependency arc a_k. The orientation of a_k depends on the state of machine k and is therefore dynamic:

Idle The machine is done processing all operations so it does not have any dependencies. Therefore the arc corresponding to this machine is absent.

Processing The continuity of this machine does not depend on other machines. In this situation arc a_k is also absent.

Blocking Suppose machine k has finished processing operation $o_{j,i}$ of job j. In order to continue processing, job j first needs to be placed on $machine(o_{j,i+1})$, the machine on which the successor operation $i+1$ of job j needs to be executed. Only by a removal of job j by $machine(o_{j,i+1})$, machine k can continue processing its own schedule. Therefore we define a blocking arc from $machine(o_{j,i})$ to $machine(o_{j,i+1})$.

Waiting Suppose machine k is waiting for operation $o_{j,i}$. Clearly operation $o_{j,i}$ is not available. The only way for operation $o_{j,i}$ to become available is when $o_{j,i-1}$, to be processed on machine $machine(o_{j,i-1})$, is finished. Therefore we define a waiting arc from $machine(o_{j,i})$ to $machine(o_{j,i-1})$.

Whenever in the process of calculating the makespan a machine state changes, the arc of the machine is updated according to its state. For every moment in time we can now derive a *Dependency Graph* that indicates for every machine what the continuity dependencies of that machine are.

Theorem 3.3.1 *If a job-shop schedule is infeasible the Dependency Graph contains a cycle.*

Proof If a schedule is infeasible, it is not possible to complete all operations. This is caused by the fact that some machines are not able to continue. If nevertheless the machines are able to continue, the operations would eventually be processed, having a feasible solution. It is easy to see that infeasible solutions can only occur if the Dependency Graph contains a cycle at some moment in time.

Assume for contradiction that we have an infeasible schedule and that the Dependency Graph graphs does not contain any cycles. Clearly, since the Dependency Graph has no cycles, it can process all operations. Completing all operations corresponds to a feasible solution, however this contradicts the assumption of an infeasible solution. Therefore if a schedule is infeasible, the Dependency Graph will contain cycles defined by dependency arcs. ∎

During the process of solving the Blocking Job Shop Problem the Dependency Graph can be maintained in order to detect infeasible situations. The following sub sections describe the various possible infeasibilities caused by cycles in the Dependency Graph:

Pure Waiting Cycles

A Pure Waiting Cycle (PWC) is caused by violations of the job constraints. This type of infeasibility is known from the Standard Job Shop Scheduling Problem. This cycle consists of only waiting arcs on the Dependency Graph, see Figure 3.3. Clearly this situation corresponds to a classic cycle in the disjunctive graph, see Figure 3.10. For the disjunctive graph it is possible to find these situations by detecting cycles.

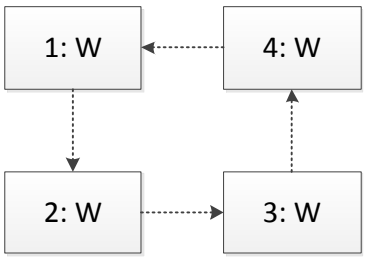


Figure 3.3: Dependency Graph of a Pure Waiting Cycle, similar to a cycle in the disjunctive graph (classical deadlock). The dashed arcs represent waiting arcs. In this figure machine 1 is waiting on an job coming from machine 2, machine 2 is waiting on a job coming from machine 3, and so on.

Pure Blocking Cycles

A Pure Blocking Cycle (PBC) consists only of blocking arcs, see Figure 3.4. The machines that are member of the cycle are finished processing and need to hand the job to the successor machine. Since the successor machine is also a member of the cycle, there is no place available for the job, therefore no machine can continue. Important in this definition is the fact that every job in the cycle comes next in the machine schedule of the successor machine. For example Figure

3.4 illustrates four blocking machines in a cycle. Each machine in the cycle has an operation to hand off to the successor machine. If we look at the individual machine schedules, we see that the next operation to be processed belongs to the same job as the machine that is blocking on this machine. Since this property holds for every machine in the cycle, the cycle is named a "Pure" Blocking Cycle.

Depending on the type of Blocking Job Shop Problem this situation is infeasible or not. For the Blocking Job Shop Problem *with swap* this situation can be resolved by simultaneously shifting the jobs to the successor machines. The Blocking Job Shop Problem *without swap* does not allow swapping the jobs, causing this specific situation to be infeasible.

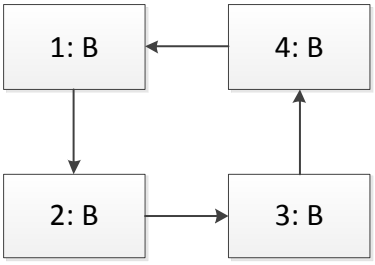


Figure 3.4: Dependency Graph of a Pure Blocking Cycle. The arcs on the cycle point to the machine where the successor operation of the job currently residing on the machine needs to go. The jobs that reside on the blocking machines are the first jobs in line to be processed on the successor machine. The solid arcs represent blocking arcs.

Blocking Cycles

The Blocking Cycle (BC) differs from the PBC in the way that there is at least one blocking arc on the cycle where machine g blocks on machine h (an arc from g to h) and the job currently on g is not the first to be processed next on machine h, see Figure 3.5. Machine g has to wait for machine h to process several several operations before machine h will remove the operation from machine g. This cycle is a derived case from the PBC.

Variable Cycles

The Variable Cycle (VC) consists of one or more blocking arcs and one or more waiting arcs, see Figure 3.6. Important to this definition is that in this cycle there always exists at least one machine that is blocking and one machine that is waiting. This cycle derived from the combination of a WC and a BC.

Complex Combinations and Emergency Buffers

As we can see in Figure 3.7, more complex types of cycles can be defined. The type of a cycle is depending on the hierarchy defined in the previous subsections. Figure 3.8 gives an overview of this cycle hierarchy. Since the simple cycle in Figure 3.7 has a waiting arc for machine 5 and other machines have blocking dependencies, the type of the cycle is labeled as a VC.

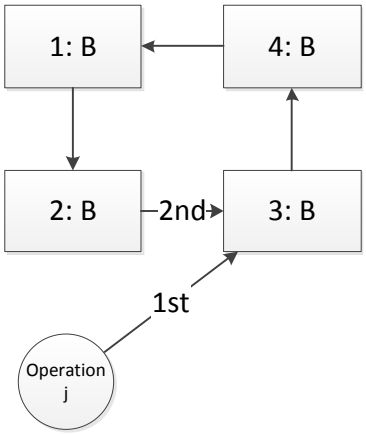


Figure 3.5: Dependency Graph of a Blocking Cycle. Machine 2 is blocking on machine 3, while the next operation to be processed on machine 3 is not coming directly from machine 2. The first operation to be processed on machine 3 is Operation j.

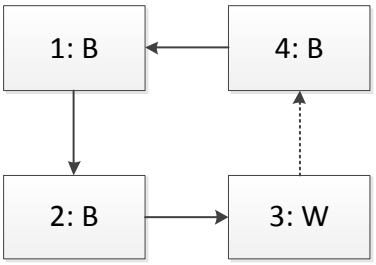


Figure 3.6: Dependency Graph of a Variable Cycle. Machine 3 is ready to process operation $O_{j,i}$, but has to wait until machine 4 has processed operation $O_{j,i-1}$. Before machine 4 can start processing operation $O_{j,i-1}$ it has to get rid of a blocking operation that must move to machine 1. Machine 1 has a blocking operation that must move machine 2 and machine 2 has a blocking operation that must move to machine 4.

The naming of the cycles in the cycle hierarchy is chosen in a way that characterizes the complexity. For example the PBC can be resolved by introducing one emergency buffer. This emergency buffer can be deployed to temporarily resolve infeasible schedules by taking out one operation to make space on a machine. Next the BC, which differs from the PBC in the way that some machine needs to wait on several operations to be processed first on the successor machine before this machine can be resolved, also consists of only blocking dependencies. Introducing an emergency buffer for the blocking dependencies in a BC can be more difficult since a suboptimal choice can be made for operations that move to the emergency buffer. Finally a VC has one or more blocking arcs and can therefore benefit from the emergency buffer, however since a VC also has waiting dependencies, the choice of emergency buffer operation is not obvious. The additional extensions, dependency arcs that not relate to the simple cycle, that arise for

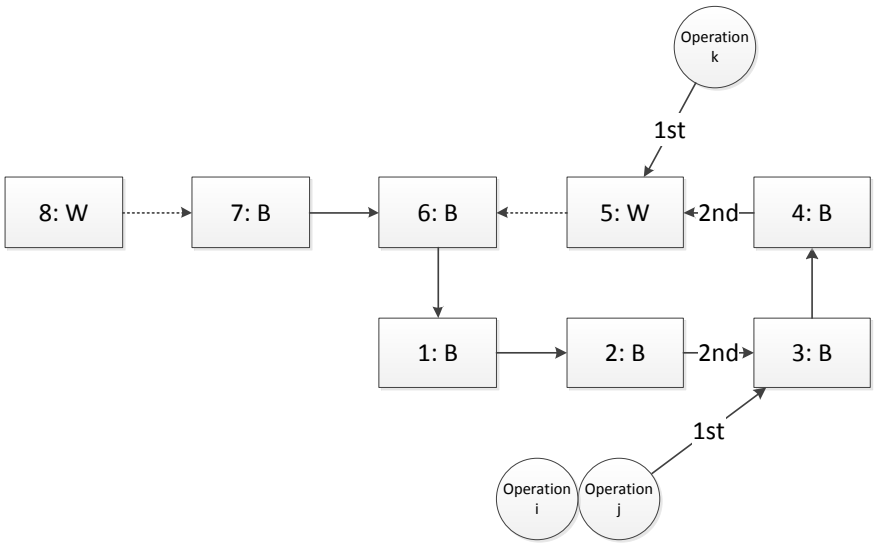


Figure 3.7: A Dependency Graph with a combination of dependency arcs. The cycle consists of dependency arcs from machine one through six.

complex combinations like in Figure 3.7, increase the difficulty for making the right decisions for the emergency buffer.

In common we can construct any type of deadlock with the Dependency Arcs. Whenever a cycle has formed, a deadlock has occurred for the machines contributing in the cycle and on the extension paths. The extension paths to the cycle do not have any influence on the type of cycle and therefore can be omitted in the analysis of the deadlock type of the cycle. Every type of cycle, with the exception of the PWC, can be resolved by deploying an emergency buffer for one of the blocking operations in the cycle. However, a dependency arc can be member of multiple cycles, therefore the choice of what operation goes to the emergency buffer is not obvious.

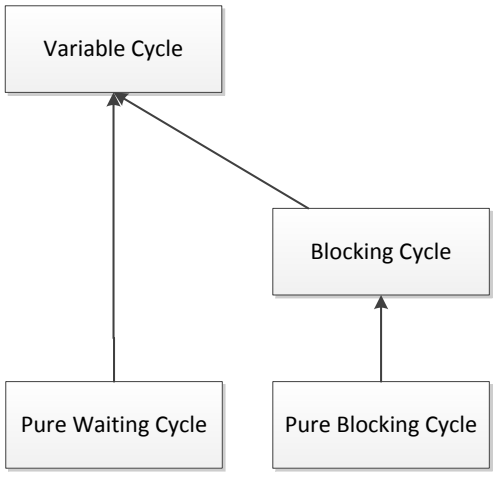


Figure 3.8: An overview of the cycle hierarchy as defined by the Dependency Graph.

3.3.2 Advanced Disjunctive Graph

In the previous section an alternative way to detect infeasible situations for the Blocking Job Shop Problem is discussed. By an analysis on the Dependency Graph different kinds of cycles can be detected that all lead to deadlock situations. However the Dependency Graph is based on a snapshot of the state of a job shop. It would be more preferable to identify the deadlock situations from a global point of view.

For the Standard Job Shop Problem the infeasible situations can be detected by the disjunctive graph. In order to detect the infeasibilities for the Blocking Job Shop Problem in the disjunctive graph we need to modify the cycle detection approach so the additional set of infeasibilities can also be detected. Indeed, by translating the Dependency Graph to the disjunctive graph this extra information can be obtained. By extending the disjunctive graph to an Advanced Disjunctive Graph the additional deadlock scenarios can also be detected. The power of the Advanced Disjunctive Graph is that it is not related to the processing times of the operations, and therefore more applicable.

Extensions to the Disjunctive Graph

In the standard formulation the disjunctive graph consists of operation vertices, job arcs and machine edges. A schedule is represented by directing all the machine edges to form a directed acyclic graph. If the resulting graph is not acyclic, an infeasible solution is obtained. However these cycles represent only the violated job constraints (classical precedence infeasibilities).

In order to detect the other types of infeasibilities, the jobs arcs are temporarily undirected, turning them into edges. This effectively enables the cycle detector to visit a broader range of vertices. The modification of the job arcs is similar to techniques used in [7]. In practice this results in undirected arcs for the job arcs, however we keep track of the original direction. To keep track of notation and context we call the disjunctive graph with the undirected job arcs the Advanced Disjunctive Graph. Finally, the cycles in the Advanced Disjunctive Graph can be mapped to cycles in the Dependency Graph.

An extra restriction on the cycle detector is that no two adjacent job arcs may be present in cycles. The cycle detector therefore only has to consider a smaller subset of vertices. Remember that the cycles are derived from the Dependency Graph and therefore correspond to continuity dependencies. Take for example Figure 3.9. In this figure two job arcs of job 2 are highlighted. If on the path of the two adjacent job arcs a cycle is formed, while the middle machine is not part of that cycle, there is no true dependency violation between the two outer operations of that job. This is because in the dependency graph the dependency arcs only exists for direct predecessor and direct successor operations. A cycle along two adjacent job arcs models a continuity dependency violation between the outer two operations, which is not relevant. Clearly if a true dependency cycle exists along two adjacent job arcs, the middle operation also has a different dependency cycle which involves only one of the outer operations. Therefore we can disregard adjacent job arcs in the cycle analysis.

Using the undirected job arcs we are able to find cycles of other types. The types we want to detect are the PBC, the BC, the PWC (classical precedence cycle) and the VC. The following sections describe how these types of cycles can be found in the Advanced Disjunctive Graph. Examples are represented by machine oriented disjunctive graphs where the horizontal arcs represent machine arcs and the vertical arcs represent job arcs. Assume for the following

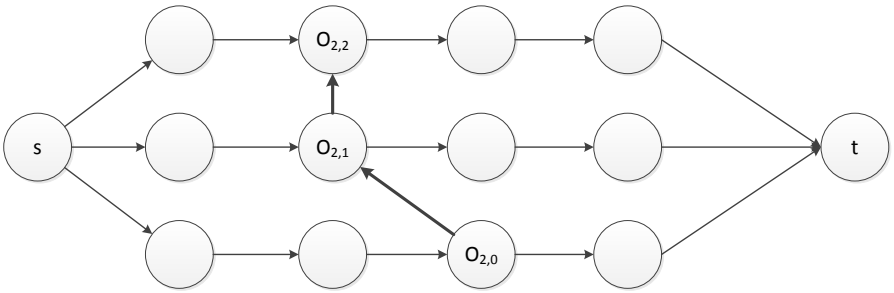


Figure 3.9: An example of two adjacent job arcs on a machine oriented disjunctive graph.

sections that whenever a cycle is given, this always represents a deadlock, which is proved later in this section.

Pure Waiting Cycles

The PWC, which is similar to the classical precedence cycle, can be identified in the classical disjunctive graph. Since the only modification is the removal of the direction for job arcs, there is no difference in detecting this cycle compared to the detection done in the disjunctive graph. The job arcs therefore need to be followed in the original direction. If we look at the Advanced Disjunctive Graph corresponding to an instance with a Pure Waiting Cycle, we see that the cycle is formed by walking across the job arcs in the original direction. See figure 3.10 for an example of the Pure Waiting Cycle.

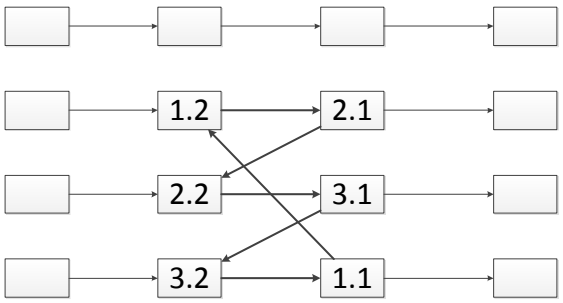


Figure 3.10: An example of a Pure Waiting Cycle in an Advanced Disjunctive Graph, formed by walking across the job arcs in the original direction.

Pure Blocking Cycles

The PBC together with the PWC are the simplest cycles to be found and have a strong relation to the Dependency Graph. Consider the dependency arcs from a PBC. Figure 3.11 consists of a set of job arcs similar to the blocking dependency arcs of a Dependency Graph. Each job arc indicates the next operation of the job, targeting the machine where it should be processed. In order for the machine to continue, the current job must be pushed along the job arc to the other machine. If we would now walk the job arcs in reverse order, making an alternating chain of

machine and job arcs, we get a cycle.

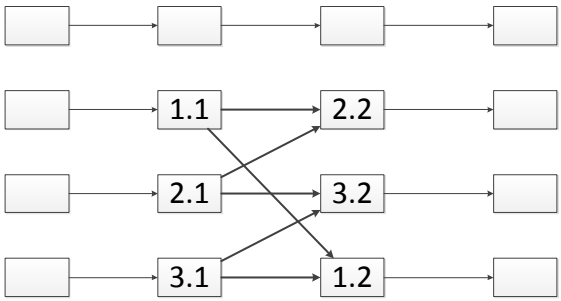


Figure 3.11: A Pure Blocking Cycle formed by walking across the job arcs in the reverse direction, with a pure alternating path of job arcs and machine arcs.

Blocking Cycles

For the BC a similar cycle to the PBC can be found. Each job arc is still walked in reverse order, however the path on the cycle does not have to be strictly alternating. From the Dependency Graph we already know that a BC is formed when one of the machines blocks on another machines while the job on which the machine blocks is not the next job to be processed on the successor machine. From an Advanced Disjunctive Graph perspective, this means that a *variable priority arc* is found. A variable priority arc is defined by a blocking arc, where the blocking operation is not scheduled next on the target machine; the target machine first has to process operations from one or more other jobs before the operation of the current job is scheduled. In this case the cycle does not consists of a purely alternating job/machine path, but more consecutive machine arcs have to be walked. The fact that more consecutive machine arcs are present in the cycle, indicates that the job that is blocking does not come first on the successor machine. See figure 3.12.

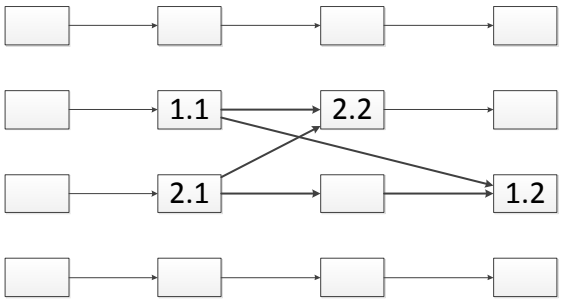


Figure 3.12: A Blocking Cycle formed by walking across the job arcs in the reverse direction, having consecutive machine arcs from operation $o_{2,1}$ to operation $o_{1,2}$.

Variable Cycle

A VC can be identified by the fact that that jobs arcs are walked in both directions. Again, no consecutive job arcs are allowed. The job arcs that are walked in reverse direction indicate

blocking dependencies. Arcs walked in the original direction indicate waiting dependencies. See figure 3.13 for an example of a VC.

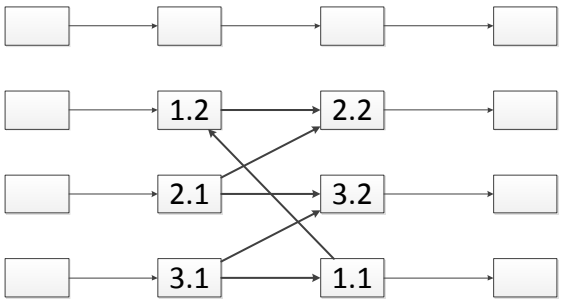


Figure 3.13: A Variable Cycle formed by walking some of the jobs arcs in the original direction (waiting) and some of the arcs in the reverse order (blocking). The arc from operation $o_{1,1}$ to operation $o_{1,2}$ indicates a waiting arc for machine $machine(o_{1,2})$; the other machines in the cycle are blocking.

3.3.3 Equivalence of cycles in the Dependency Graph and the Advanced Disjunctive Graph

The infeasibility cycles described for the Dependency Graph and the Advanced Disjunctive Graph are equivalent. In order to prove this, we look at how we can transform a cycle from the Advanced Disjunctive Graph into a cycle in the Dependency Graph and vice versa. First consider the relation between a Dependency Graph and an Advanced Disjunctive Graph. We already know that the Advanced Disjunctive Graph is a global overview of the job shop instance. The Dependency Graph on the other hand is snapshot of a specific state. Therefore it is to nobody's surprise that a Dependency Graph is only related to subgraph of the Advanced Disjunctive Graph.

Theorem 3.3.2 (Construction of Dependency Graph from Advanced Disjunctive Graph) *If an infeasibility cycle exists in the Advanced Disjunctive Graph, an equivalent infeasibility cycle exists in the Dependency Graph.*

Proof Assume we have an Advanced Disjunctive Graph with a cycle. For each machine k we construct a corresponding vertex V_k. Next for each machine in the cycle we identify the type of arc that corresponds to the machine state. Because the machine arcs in the Advanced Disjunctive Graph can only be walked in one direction, we can for each machine identify a *head* and a *tail*. The operation where the cycle enters the machine is called the head, the operation where the cycle leaves the machine is called the tail, see Figure 3.14.

We can now identify the state for the current machine. Since we assumed this machine is part of the cycle, the cycle must visit the head and the tail vertices. Because the machine can not continue, the machine is stuck at the head vertex. The reason why the machine is stuck at the head vertex is because the head has a dependency relation on a tail vertex of another machine. Since tail vertices correspond to operations which are unreachable, the head vertex is also unable to continue. The incoming job arc for the head vertex indicates the dependency

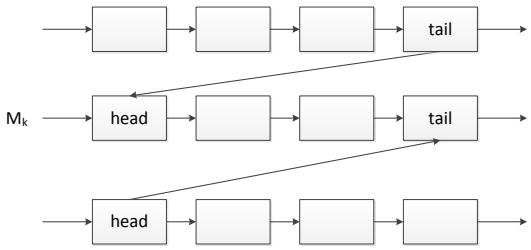


Figure 3.14: Example of a head and tail operation on a machine oriented Advanced Disjunctive Graph.

relation and explains why this machine can not continue. In Figure 3.14 the incoming job arc consists of a job arc in the original direction. Therefore we know that the dependency arc is a waiting dependency.

Finally in order to form a cycle in the Advanced Disjunctive Graph, the cycles must consist of alternating tail / head connections, because the machine arcs force a direction on the cycle. Since every machine in the cycle has a head vertex that is dependent on the tail vertex of another machine, the schedules can not continue. By collapsing the head and tail vertices in the Advanced Disjunctive Graph, we get an equivalent cycle for the Dependency Graph, in which the tail / head connections indicate the type of dependency arcs. ∎

Theorem 3.3.3 (Construction of Advanced Disjunctive Graph from Dependency Graph) *If an infeasibility cycle exists in the Dependency Graph, an equivalent infeasibility cycle exists in the Advanced Disjunctive Graph.*

Proof In a similar way we can construct an Advanced Disjunctive Graph from a Dependency Graph. Assume we have a Dependency Graph with a cycle. For each machine that is member of the cycle in the Dependency Graph, we identify the current operation. Whether the machine is waiting on the current operation is or blocking on that operation, indicates the head-tail relation in the Advanced Disjunctive Graph. Since the machine is part of the cycle in the Dependency Graph, we can check the machine that has a dependency on our machine. Because the machine from the incoming dependency arc can not continue (it is is a member of the cycle), it must be depending on an operation that comes from the current machine that is not yet executed. This indicates that the machine is targeting an operation on the machine schedule later than the head operation, otherwise it would have been available. If the dependency arc is a waiting arc, the orientation of the head-tail relation is directed to the head. In the case the dependency arc is a blocking arc, the head-tail relation is directed to the tail.

For each machine in the Dependency Graph cycle we can already identify the head operation in the Advanced Disjunctive Graph. Finally since there is a deadlock, the machines are waiting or blocking on an operations that comes later on schedules of other machines in the cycles. By following the dependency arcs in the cycle for the Dependency Graph, we define the head tail relations in the cycle for the Advanced Disjunctive Graph. ∎

Corollary 3.3.4 *The infeasibility cycles in the Dependency Graph and the Advanced Disjunctive Graph are equivalent.*

Proof By combining the implications from Theorem 3.3.2 and Theorem 3.3.3 a bi-implication is formed, proofing the equivalency relation between the infeasibility cycles in the Advanced Disjunctive Graph and Dependency Graph. ∎

3.4 Solution Approaches

In this section we give a detailed overview of existing solution approaches in the literature. Next to the existing solution approaches in the literature, we propose two newly developed methods. Also a new approach for calculating the objective function is proposed. The advantage of this new approach is that it improves the worst case runtime complexity. It also identifies the infeasibilities from the result of Section 3.3. Section 3.5 presents computational results from the solution approaches described in this section.

3.4.1 Mascis and Pacciarelli [16]

In [16] Mascis and Pacciarelli developed an *alternative graph* to model the blocking and eventually no-wait constraints for the Blocking Job Shop Problem and No-Wait Job Shop Scheduling Problem. The No-Wait Job Shop Scheduling Problem $(J \mid no\text{-}wait \mid C_{max})$ is an extension of the Standard Job Shop Scheduling Problem, with the additional constraint set in which each successor operation of a job should immediately be started after finishing the predecessor operation. A solution to a job shop problem is expressed as a complete selection of the disjunctive arcs in the alternative graph. In a complete selection for each disjunction a single arc from the pair is selected. If the selection yields a directed acyclic graph, the selection is a complete consistent selection and the solution is feasible. Otherwise the solution is infeasible.

In their research four greedy heuristics are developed that gradually increase the selection of disjunctive arcs in a partial schedule. An important result is the fact that the problem of extending a partial selection to a complete consistent selection is NP-Hard. It is therefore possible that in the process of solving a Blocking Job Shop instance using their heuristics, an infeasible – non consistent – solution is obtained.

In each iteration a single disjunctive arc in the alternative graph is selected. The next disjunctive pair, for which an arc has to be selected, is determined by one of the following heuristics:

- Avoid Maximum Current C_{max} (AMCC)
- Select Most Critical Pair (SMCP)
- Select Most Balanced Pair (SMBP)
- Select Max Sum Pair (SMSP)

For the AMCC heuristic first the disjunctive arc is found that, when selected, results in the maximum makespan. This is done by iteratively fixing unfixed disjunctions and calculating the makespan. Next the complement arc, from the disjunction that causes the largest makespan, is selected to avoid leading to this maximum makespan.

The SMCP heuristic first determines which pair is the most critical pair, and selects the arc from the pair that results in the smallest makespan. For each disjunctive pair we again

calculate the makespan if one of the arcs is selected. Then for each pair the minimal makespan is determined. The pair that has the maximum 'minimal makespan' is considered to be the most critical pair. The disjunction that results in the smallest makespan is selected.

The SMBP heuristic calculates for each unselected disjunctive pair the difference in makespan if either of the arcs is selected. Next from the pair that results in the smallest difference in makespan between the two disjunction, i.e. the pair that is most balanced, the arc is selected that results in the smallest makespan.

Finally the SMSP is similar to the SMBP, however instead of determining the difference in makespan for each arc in the disjunctive pair, the sum is calculated. From the pair that results in the maximum sum for the two makespans, the arc is selected that results in the smallest makespan.

From all results, the best results for the Blocking Job Shop Problem *without swap* are taken for comparison against other solution approaches. Column MP in Table 3.2 describes these results.

3.4.2 Gröflin and Klinkert [8]

The approach taken by Gröflin and Klinkert [8] consists of a tabu-search algorithm with a specialized neighbourhood. In contrast to the solution approach taken by Mascis and Pacciarelli, this approach remains feasible in every step of the algorithm. As indicated in their research, a simple swap neighbourhood for the Blocking Job Shop Problem based on critical pairs, often results in infeasible solutions.

Based on previous research results [7], Gröflin and Klinkert developed a theoretical foundation on inserting multiple operations in a job shop. Using this theory they worked on a tabu-search algorithm that in each iteration inserts an entire job. First, the critical path of a solution is determined. On this critical path a disjunctive arc is chosen from which either one job is chosen. In the chosen disjunctive pair, the complement disjunctive arc is fixed and the job is removed from the problem to be reinserted subsequently. While the job is inserted again, all disjunctions pairs related to this job are unfixed to choose the disjunctive arcs that result in the minimal makespan.

The power of this method lies in the fact that the neighbourhood potential is much larger than when only considering single swaps. Also in the process of reinserting whole jobs, only feasible solutions are allowed. While reinserting a job, the selection of disjunctive arcs corresponding to this jobs is iteratively increased. In each iteration a check is done whether the solution is still feasible. Using the previous results on short cycles [7], this check can be done efficiently. The results from this research are used in comparison for the Blocking Job Shop Problem *with swap*.

3.4.3 Van den Broek [26]

In [26], Van den Broek developed a Job Insertion Heuristic (JIH) that reinserts jobs similar to the algorithm developed by Gröflin and Klinkert. Instead of creating a customized algorithm to reinsert jobs, his heuristic relies on the perofmance of a MIP. The MIP formulation explained in Section 3.2.1 is based on the MIP formulation described in his paper.

The heuristic starts with an empty solution in which no jobs are present. In the first phase the jobs are iteratively inserted according to the largest total processing time. For each job a

Job Insertion Problem (JIP) is solved. The JIP is formulated by a MIP formulation in which for the new job all the decision variables are unfixed, while the other decision variables remain fixed. Next when all jobs are inserted by the JIP, a reoptimization step is applied where jobs are being reinserted until no improvement can be found anymore. A reoptimization steps starts by determining the operation with the maximum processing time, and reinserting the corresponding job by solving the JIP for that job. Next this process is repeated until no further improvement can be found.

Interesting in this approach is that after the initialization phase, the solution obtained is always feasible. In the worst case scenario, the job that is being inserted is placed at the end of the schedule. It is also easy to see that in the optimization phase, the solver will remain finding feasible solutions. This is due to the fact that the solver will always start with feasible solutions, therefore it could always fall back on the initial solution.

The power of the JIH is the fact that the JIP sub problems are fairly easy to solve. This is because when inserting a job, the decision variables not related to the job are fixed in the MIP. As a result, the solver only has to determine a small amount of decision variables. Also in the branch and bound phase of the solver, infeasibilites are easiliy detected, rapidly leading to better solutions.

3.4.4 Simulated Schedule Observer

In our research on the Blocking Job Shop Scheduling Problem a new method for determining the objective function is developed. Similar to the Stochastic Job Shop Problem, we focus on the minimization of the *makespan*. Since the Blocking Job Shop Problem is subject to a broader range of infeasibilities than the Job Shop Scheduling, a new approach for calculating the objective function and evaluating the feasibility is developed. The main advantage of this approach is that only a single method is required to: determine the feasibility status, get feedback information on partial infeasible solutions and the calculation of the actual makespan.

The challenge in the development of an objective function that is able to supply the previous described information, is to optimize the runtime of the algorithm, as the objective function is executed whenever the objective value needs to be determined. Therefore it is important to minimize the worst case runtime.

Inspired by the runtime execution of the discrete-event simulation for the Stochastic Job Shop, a new approach to calculate the objective function is explained in this section.

Algorithm Runtime Details

The algorithm to calculate the objective function is closely related to the execution of a discrete event simulation. From the discrete even simulation used in the Chapter 2, we already observed a runtime execution which requires for each operation an event to be scheduled. Furthermore, each event takes a constant amount of computation time and needs to be sorted in increasing order of timestamp. To summarize the runtime analysis utilizing a sorting algorithm that takes $O(log(n))$ time, the execution of the discrete event simulation requires $O(n \cdot log(n))$ time, where n is determined by the total amount of operations in the Job Shop.

The advantage of utilizing a discrete event simulation is the fact that whenever a schedule is infeasible, the simulation stops generating events and returns prematurely. The fact that the simulation stops allows the observer to identify where the infeasibilities come from. One dis-

advantage is that whenever the deadlock situations occur the simulation observer has advanced to the latest timestamp. Reason for this disadvantage is the fact that when a deadlock occurs, all the machines that participate in the deadlock have already spent time waiting or blocking. When these deadlocks are temporarily resolved, we effectively ignore the infeasibility, but the machine timestamp has a small error since the deadlock is resolved at a worst case moment in time and the machines corresponding to the deadlock already spent time blocking or waiting.

The execution of a discrete event simulation can be logically imitated without the need of a sorted event list to determine the order of events. In fact, the objective function can be determined without the need of a global simulation clock to keep the entities and events synchronized. First consider the logical flow for an operation of a job, described in Figure 3.15.

Figure 3.15: The three phases a job shop operation can possibly go through

Figure 3.15 summarizes the three distinct phases each operation goes through. Since for the Blocking Job Shop preemption is not allowed and the fact that each machine can only process one operation simultaneously, it follows that a machine also follows the same phases transitions. In fact, the phases described in the figure illustrate a worst case scenario. If for example an operation is already marked available, the *waiting* phase is omitted. The same reasoning applies for the *blocking* phase, if an operation is finished and the subsequent machine is already waiting for the operation, the *blocking* phase is also omitted.

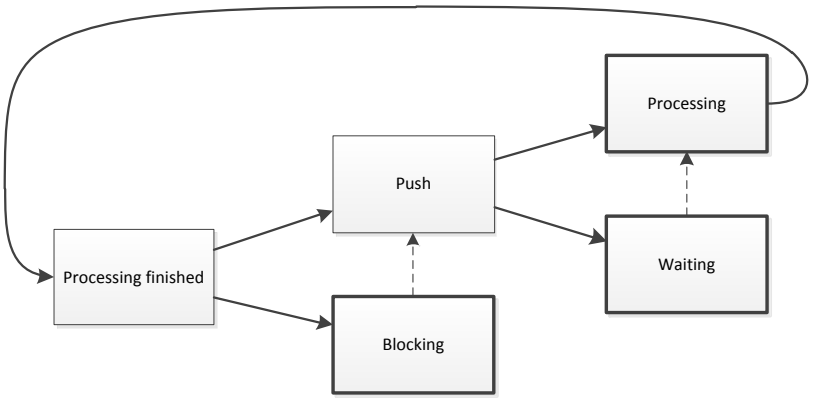


Figure 3.16: The workflow graph of a machine. The highlighted elements relate to the three distinct phases from Figure 3.15.

From the machine workflow graph in Figure 3.16 we can derive a new algorithm that performs these state transitions. Suppose that for each machine state (*waiting*, *processing* and *blocking*)

we construct a list, respectively a *WaitingList*, *ProcessingList* and a *BlockingList*. Each machine, if not finished processing, resides in one of these lists. The algorithm, which is called *Simulation Schedule Observer*, now keeps processing machines from the processing list.

The SSO algorithm is initialized by first determining the initial state of the machines. If the first operation of a machine schedule is available, the machine is put in the *ProcessingList*, otherwise the machine is put in the *WaitingList*.

```
method SimulationScheduleObserver()
{
        //while there are machine to process
        while (ProcessingList is not empty)
        {
                currentmachine = ProcessingList.Dequeue;

                ProcessMachine(currentmachine);

                if (PushOperation(currentmachine))
                        PullOperation(currentmachine);
        }
}
```

Listing 3.1: Pseudo code of the Simulation Schedule Observer

Since the Simulation Schedule Observer does not schedule individual events, there is no need for a global simulation clock. Instead, each machine m keeps track of its individual machine time t_m. The advantage of this approach is that the required sorting time for the event list is omitted, dropping an $O(log(n))$ factor from the worst case runtime. However, in order to have a valid objective function, we still require synchronization steps between machines.

If we look at the pseudo code listed in Listing 3.1 we can identify two state transition. The *PushOperation* and *PullOperation* synchronize the machine times and model a state transition depending on the result of the synchronization. For completeness we will describe the steps involved in synchronizing the machines. Note that the state transitions are applied by synchronizing the machines, therefore the order of the *ProcessingList* is unimportant.

PushOperation:

1. If the operation of the current machine is the last operation of its job, the push is actually release the job from the problem, therefore this action can always continue and returns *true*.

2. If the successor machine of the current job is not yet waiting for the current job, the current machine can not continue. Therefore the machine will be appended to the blocking list. And the function returns a *false*.

3. If the successor machine is waiting for the current job we apply a synchronization between the two machines. Next the successor machine is appended to the *ProcessingList* and we try to pull the next operation from the schedule of the current machine. The return in this case is *true*.

PullOperation:

1. If the current machine is finished processing all operations the machine is removed from all lists.

2. If the successor job of the current machine is not yet available on a machine, because the predecessor operation still needs to be processed by another machine, the current machine is appended to the WaitingList.

3. If the successor job of the current machine is available on a blocking machine, the blocking machine is released and the machines are synchronized. Next the current machine is appended to the ProcessingList. Finally the machine where the current job originates from will recursively apply a PullOperation to check whether that machines can also continue.

In the previously described steps we can see that the machine times do not play a role in the determination whether a job is available or not. From the description we can also see that the synchronization step is always applied between a pushing machine (that is possibly blocking) and a pulling machine (that is possibly waiting). After the machines have been processed the machines times are increased by the length of the processing time of the current operation. Next if two machines, for instance blocking machine m_b and waiting machine m_w, are synchronized, the difference between the machine times are calculated. The difference in machine times, $t_b - t_w = \Delta_t$, has three scenarios:

- $\Delta_t = 0$: If the difference between the machine times is exactly zero, the machines were finished processing at the same moment. In this case the blocking machine did not spend any blocking time and the waiting machine did not spend any time waiting on the job. See Figure 3.17a.

- $\Delta_t > 0$: If the difference is positive, the blocking machine was finished later than the waiting machine. The machine time of the waiting machine is increased by Δ_t and the same amount of waiting time $w(O_{j,i})$ is recorded for the specific operation $O_{j,i}$. In this case the blocking machine did not spend any actual time on blocking; $b(O_{j,i-1}) = 0$. See Figure 3.17b.

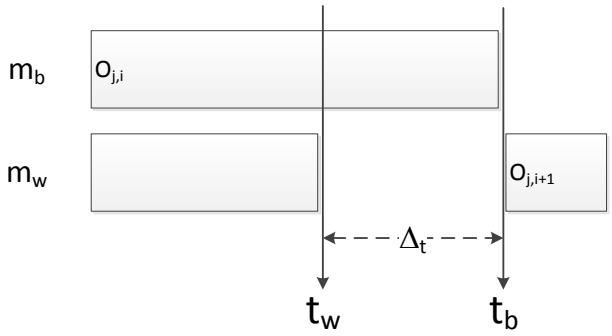
- $\Delta_t < 0$: If the difference is negative, the waiting machine was finished later than the blocking machine. The operation currently residing on the blocking machine receives a blocking time $b(O_{j,i})$ equal to $-\Delta_t$ for the specific operation $O_{j,i}$ and the machine time is increased by $-\Delta_t$. For this case the waiting machine will not spend time waiting on the job. See Figure 3.17c.

Deadlock Detection

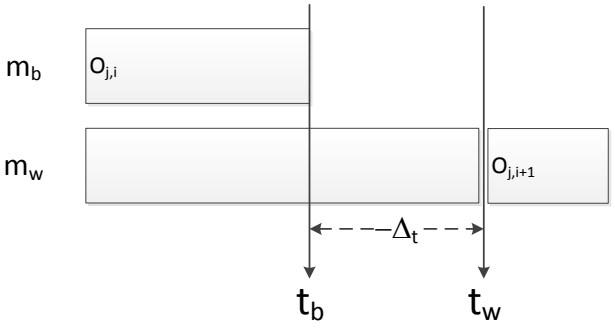
During the execution of the Simulation Schedule Observer, the machines from the ProcessingList activate machines from the WaitingList. Also the PullOperation method recursively activates machines from the BlockingList by the PushOperation. If the objective function is run for a schedule without deadlocks, all the machines will eventually finish processing all operations, see Theorem 3.3.1 for a similar argument. However if a schedule contains deadlocks, it is impossible to complete all the machine schedules.

Whenever the ProcessingList contains machines, it is possible to advance the schedule calculation. Since only machines from the ProcessingList can activate machines from the WaitingList

(a) Example of a synchronization between machines that have the same timestamp. In this case the waiting machine m_w does not spend any waiting time and the blocking machine m_b does not spend any time on blocking.



(b) Example of a synchronization between machines where the timestamp of the blocking machine m_b is larger. In this case machine m_w spends Δ_t time on waiting.



(c) Example of a synchronization between machines where the timestamp of the waiting machine m_w is larger. In this case machine m_b spends $-\Delta_t$ time on blocking.

Figure 3.17: Examples of machine synchronizations between a blocking machine m_b and a waiting machine m_w.

and the BlockingList, the schedule can be regarded infeasible if the ProcessingList is empty and not all jobs are finished processing. The advantage of the Simulation Schedule Observer is that the types of deadlocks from Section 3.3 can easily be detected. At the moment a deadlock occurs, the deadlock type is determined and the caller function is informed of the type of deadlock. If the deadlock contains a cycle with blocking machines, the caller can choose to temporarily release a blocking machine by placing the blocking operation in a fictive emergency buffer. Since the machines keep track of individual machine times and are not yet synchronized, we are able to solve the deadlocks prematurely. Depending on the machine that is released, the schedule calculation receives only a small error regarding the calculation of start times and end times. Because the machines also record the operation the machine is waiting for or blocking on, we have instant access to the Dependency Graph.

Runtime Analysis

In each definition of a job shop, we have that whenever a solution is feasible, all operations will be processed. Similar if all operations can be processed the solution is feasible. By a counting argument it is easy to see why the SSO algorithm runs in $O(n)$ time for feasible instances.

Theorem 3.4.1 (Linear Simulated Schedule Observer) *The Simulated Schedule Observer algorithm runs in $O(n)$ time on feasible instances.*

Proof Consider machine m and its state transitions for a single operation o. Assume that the job-shop schedule is feasible.

Before the machine can start processing o, it has to wait for operation o to become available. The machine is therefore put in the *waiting list*. If the operation was available at that specific moment in time, the machine would be able to continue and be placed in the *processing list*. In the case the operation is not available, a *blocking* machine would eventually activate the machine.

Next the machine is removed from the processing list and the machine state is updated. The next action for the machine is to push the job to the successor machine. If the successor machine is not waiting for the operation, machine m is put in the *blocking list*. When in the blocking list, the machine has to wait for the successor machine to remove the job.

In the best case scenario when working on operation o, machine m can immediately start processing, thereby skipping the waiting list. And when finished the machine can immediately push the job to the successor machine, skipping the blocking list. For the worst case scenario, machine m goes through the three lists for each operation. Because we have assumed that the schedule is feasible, we know that every operation will be processed. Also for each operation a machine will for the worst case scenario go through three states. Since each state transition is applied in constant time, we come to a worst case runtime of $3n$ which is equivalent to $O(n)$, where n is defined by the total amount of operations. ∎

Important in the above proof is the fact that all operations will be processed at some point in time. This is due to the fact that blocking machines are always eventually released by a waiting machine and vice versa. For the Blocking Job Shop Problems *with swap* this is however not always the case, since for this type of job shop, a cycle of pure blocking machines is allowed. The PBC results in a set of machines that all reside in the blocking list. However, due to the special structure of a PBC, these cycles can be detected.

Theorem 3.4.2 (Linear Simulated Schedule Observer for PBCs) *The Simulated Schedule Observer algorithm runs in $O(n)$ time on instances with Pure Blocking Cycles.*

Proof By tracking additional problem information, cycles of the type PBC can be detected and automatically resolved. During the state transitions additional pointers need to be tracked for machines that enter a blocking state. Whenever a machine is pure blocking, that is the job currently on the machine needs to be processed next on the other machine, this information is recorded. Additional pointers can keep track of chains consisting of pure blocking machines, that eventually can form a Pure Blocking Cycle. If the final machine in a chain of pure blocking machines completes the cycle, the machine with the highest machine time can be resolved, allowing all machines in the cycle to continue. Keeping track of the pure blocking chains, the machine pointers and the machine in the chain with the highest processing time can be done in constant time. Therefore a PBC can be resolved in constant time, resulting in a total runtime complexity of $O(n)$ time, for instances with Pure Blocking Cycles. The time-stamp of a Pure Blocking Cycles that is released, is therefore always equal to the machine with the highest time-stamp; the machine that is last finished processing. ∎

Finally, the other infeasibilities can also be detected. However, the type of the infeasibility needs to be determined by an analysis on the cycle. This analysis requires extra computing time to complete.

Theorem 3.4.3 (Simulated Schedule Observer for k infeasibilities) *The Simulated Schedule Observer algorithm runs in $O(n + k \cdot m)$ time on instances with k infeasibilities.*

Proof Consider a job shop instance with one infeasibility. By the cycle analysis given in Section 3.3, each infeasibility corresponds to a cycle in the dependency graph. When calculating such a schedule in the SSO, eventually the processing list will be empty while not all operations are processed. Each machine that corresponds to the infeasibility resides in either the blocking list or the waiting list. The type of infeasibility can be determined by calculating the Dependency Graph from the current SSO state. Since determining the state for a single machine can be done in constant time and the state of all machines need to be determined, this requires m time.

If for instance a BC, a PBC or a WC is released by an emergency buffer, more infeasibilites can occur at later stages in the schedule. For each of these infeasibilities the type of cycle is determined by the analysis. Therefore, a schedule with k infeasibilities requires $k \cdot m$ time on analyzing the type of infeasibility. Together with the linear runtime of the SSO, the worst case runtime for a schedule with k infeasibilities requires $O(n + k \cdot m)$ time. Note that the amount of infeasibilities, k, also depends on what operations are released to the emergency buffer, since several operations can correspond to multiple infeasibilities. ∎

3.4.5 Random Inserts

The first approach for solving the Blocking Job Shop Problem, is by using the MIP formulation in a hill climber. The full formulation of the MIP requires too much computing power to solve the complete problem. Therefore when applying the MIP we fix a certain combination of variables. In the process of solving the Blocking Job Shop Problem we used the The Gurobi 5.0 optimizer [9]. The advantage of this solver is its ability to detect constraints that are superfluous so when applying the solver to a fixed instance, a large amount of rows and columns are automatically

removed. Also this solver is, used a lot in scientific research, and is known for its remarkable results.

Similar to the JIP in the JIH from Van Den Broek, we have developed a MIP that fixes a partial solution and reinserts a specific set of operations. First, the algorithm starts with a worst case feasible solution where all jobs are scheduled as a flow-shop. In this solution each machine processes the jobs in the same order. The results of this order is that the machines have minimal mutual activity. Next the general idea is to iteratively reinsert a selected subset of the operations. For the selection of the operations the following two heuristics are used:

- *Random Inserts.* For the current solution, three random jobs are chosen. For these jobs all operations are selected. Next the set of selected operations is extended with all the operations of two random machines.
- *Random Critical Inserts.* For the current solution, three random jobs *from the critical path* are selected. For these jobs all operations are selected. Next the set of selected operations is extended with all the operations of two random machines.

The critical path is defined by the longest path in the schedule; this is the path that determines the makespan. A consequence of delaying operations on the critical path is the increase of makespan.

First note that in the selection of the operations from the jobs and the machines, there could be an overlap in operations. For each operation in the selected set, the disjunctive decision variables in the MIP are unfixed. Since in our case each machine processes J operations, this results in $J-1$ pairs of decision variables that are unfixed. The decision variables related to the other operations are fixed.

As in [26] this approach will result in much smaller problems. More important is the fact that this approach uses a large neighbourhood. As Gröflin and Klinkert already stated, a neighbourhood of simple swaps does not produce good or even feasible solutions. The reinsertion of whole jobs proves to be an important improvement. To take this improvement further, the Random Insert heuristic combines several jobs and machines to be reinserted. The possible interaction between the operations in the set being reinserted can now be taken into account, resulting in a larger neighbourhood that is able to explore more solutions. Similar to the MIP formulation used by Van den Broek, the formulation will always result in feasible solutions.

When the Random Insertion approach is applied multiple times, each iteration the solver is allowed to find improving solutions. Since the neighbourhood is randomized, the solver will less likely run into local optima. Finally because the solver will never find deteriorating solutions, the algorithm is equivalent to a hill-climber algorithm with a limited amount of iterations.

3.4.6 Extended Random Inserts

An improvement to the Random Inserts solution approach, is the Extended Random Inserts. By allowing the Random Insertion hill climber to do multiple restarts, the Extended Random Inserts method is able to do a broader search in the landscape of solutions.

First a worst case solution is initialized in which all machines have the same job orders. This results in a schedule where there is least amount of mutual activity between the machines. Next a single run of the Random Insertion hill climber is applied to obtain a current best solution. In

the following phase the Random Insertion hill climber is restarted multiple times on modified schedules.

The modification of the schedule is done by loading the current best solution and applying random swaps. The random swaps consists of randomly applying all possible swaps. Since this often results in infeasible schedules, we use the SSO algorithm to determine after each swap whether the resulting schedule is still feasible.

At the end of a fixed amount of Random Insert hill climber iterations on modified schedules, the best found schedule is returned as final answer.

An important element in this approach is the fact that in the phase of the multiple restarts, the Random Insertion hill climber starts with a solution that is much better than the worst case solution. In this way, the solver is able to explore more regions, since the initial iterations of optimizing the worst case solution can now be used for further optimizing better solutions. Also in the case the solver results in a local optimum, the restart on a modified schedule will sometimes force the solver to focus on different parts of the solution.

Compared to the iterations of the Random Insertion hill climber, the iterations of the Extended Random Insertion method will not always improve the current best schedule. This is due to the fact that the Random Insertion iterations will sometimes find different local optima that are worse than the current best solution.

3.4.7 Local Search With Emergency Buffers

In this subsection we will propose a hypothetical solution approach based on the usage of an emergency buffer. As the previous results already showed, the neighbourhood for altering intermediate solutions is very important. Also the classical swaps that work good for the Standard Job Shop Scheduling Problem cannot be applied to the Blocking Job Shop Problem since most of the time this results in infeasible solutions. The solution approach discussed in this subsection will overcome this problem by temporarily ignoring a specific category of infeasibilities.

Suppose that for some practical problems an emergency buffer or emergency storage location is available. If the practical environment is related to the Blocking Job Shop Problem, the absence of unlimited intermediate buffers plays an important role. By allowing some operations to go into the emergency buffer, the process is able to continue, and since this reduces the amount of blocking time the makespan is also reduced.

The SSO algorithm gives us the ability to access detailed information about infeasible situations. Especially the blocking scenarios are interesting since these can be resolved by the emergency buffer. Also the MIP formulation can be initialized in such a way that only a specific part of the schedule is being optimized.

When solving the Blocking Job Shop we can thus apply a local search algorithm that applies the standard swaps, and temporarily allow a violation of the blocking constraints. In the process of applying the local search, the amount of emergency buffer usage can be taken into account by the objective function. Note that the decision of what operation needs to go into the emergency buffer is not trivial. Next the local search phase can be alternated with a repair phase in which the MIP formulation can be applied to eliminate the infeasibilities.

In the process of 'repairing' cycles, it is important to focus the attention on only a small subset of the problems. If for instance in the local search process multiple emergency buffers are used, we should try to remove each emergency buffer individually. It is easy to see that each

emergency buffer is a result of a specific cycle. When calculating the objective value for the final schedule, we could therefore record all the cycles that occurred in the Dependency Graph. Next the repair process is started on the cycle that occurred last.

In order to solve a single cycle, the MIP formulation must be initialized in a specific way. Figure 3.18 gives an example of a machine oriented solution where the latest cycle is marked by the dark grey elements. Since the marked cycle is the latest cycle that exists in the schedule, we know for sure that the operations from Section C do not contain any infeasibilities. In order to execute the solver to solve the marked cycle, we must fix all the machine indexes for the operations in Section A. Furthermore since Section A could contain additional cycles, the blocking constraints corresponding to this section should therefore be ignored. Next to keep the solver closely related to the result found by the SSO, the start times of the last operations in Section A should also be fixed by the start times found in the SSO. Finally the operations from Section B, the operations that cause the cycle, should be unfixed in order for the cycle to find new machine positions.

Since it has already been proved that extending a consistent selection to form a complete consistent selection is NP-Hard, it will most probably be difficult to solve the cycles for a partial schedule [16]. Therefore in order to support the solver, additional operations from Section C should also be unfixed. Heuristics for selecting these operations still need to be developed.

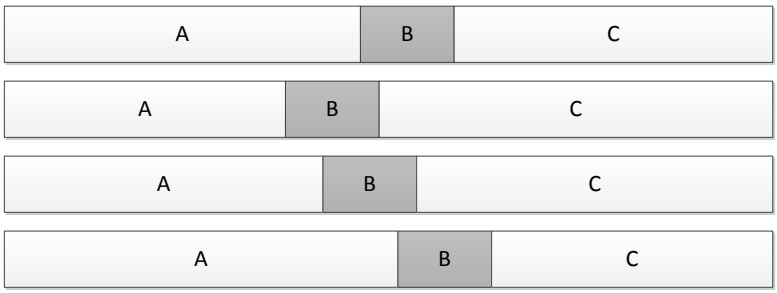


Figure 3.18: Example of a machine oriented solution. The dark grey elements correspond to the set of operations that form the latest cycle in the Dependency Graph. Operations in Section A could also form cycles in the Dependency Graph, operations from Section C do not contain any infeasibilities.

3.4.8 Local Search With Fixed Swaps

Similar to the approach of Gröflin and Klinkert [8] we can apply a local search in which we invert disjunctions to obtain neighbour solutions. In their research, one of the incident jobs of the disjunction gets reinserted. Additionally instead of reinserting the job using a custom algorithm, we could also apply the MIP formulation. By fixing the inverted disjunction and unfixing all operations for either one of the incident jobs, an equivalent functionality is obtained.

Extending the amount of unfixed operations increases the search space for the MIP solver. Instead of reinserting only one incident job, both jobs can be reinserted, while the swap remains fixed. Similar to the Random Insertion methods, an extension can be added to the selection of unfixed operations to give the solver more freedom.

However during experiments with this approach, the fixed swaps forces the solver to sometimes search in a suboptimal solution space. Also extending the unfixed operations only led to an increase of computing time without improving the solution. This approach is similar to the Random Insertion method, however the swap forces specific solutions. As the results in Table 3.1 show, the Random Insertion methods prove to be very effective, therefore this approach is not further investigated in this thesis.

3.5 Results and Analysis

The Random Insertion and the Random Critical Insertion are compared with known results from the literature. The following list of Job Shop instances, which are digitally available on the web [3], are selected:

- *Abz5–9* have been generated by Adamt et al. [1].
- *Orb1–10* have been generated by Applegate and Cook [2].
- *La1–40* have been generated by Lawrence [13].
- *Ft6,10 and 20* have been generated by Fisher and Thompson [17].

Table 3.1 on page 78 shows the results of applying the Random Insert and the Random Critical Insert methods for the Blocking Job Shop Problem *with swaps*. Table 3.2 on page 79 shows the results of applying the Random Insert and Random Critical Insert method for the Blocking Job Shop Problem *without swaps*. For each experiment the amount of insertion iterations is fixed to 150. Since the solution approaches depend on randomization, the results vary per execution of the algorithm. Therefore we repeat each experiment five times, from which the minimum and the average final makespans are listed in the tables.

As the results in the tables show, the Random Insert and Random Critical Insert solution approaches result in improvements for most of the instances. Interesting to note is that for several instances the optimum is found by our insertion methods. However, for some instances, the distance to the optimal solution is still large. For the larger instances, starting at *La21*, the Random Insert and Random Critical Insert methods perform better than the results from the literature.

The Extended Random Insertion method is applied to the same set of instances. After the initial optimization of the worst-case schedule, 9 iterations are applied in which the current best solution is modified and reoptimized. The results of the *Extended Random Insertion* and *Extended Random Critical Insertion* are also listed in Table 3.1 for the Blocking Job Shop Problem with swap, and Table 3.2 for the Blocking Job Shop Problem without swap.

Along with the final minimum makespan, also the initial (worst-case) makespan is listed. Column $\overline{\text{Avg Pre-Opt}}$ lists the average makespan of the 9 modified solutions before the Random Insertions are applied. The initial makespan is not taken into account for the $\overline{\text{Avg Pre-Opt}}$ performance measure. The column $\overline{\text{Avg Post-Opt}}$ lists the average makespan of 10 iterations of Random (Critical) Insertions.

Both solution approaches result in improvements for most instances. Especially for large instances in which the number of jobs is 20 and the number of machines equals 10, the solutions

are better. However, for these instances the insertion solver takes considerably more time than for the smaller instances. In some case, the branch and bound algorithm of the MIP formulation requires more than 15 seconds to complete, which is long compared to an average of less than 2 seconds.

Interesting is the fact that the Extended Random Insertion methods sometimes result in solutions that are worse than the solution produced by the normal Random Insertion methods. A reason for this behaviour could be the fact that after the initial optimization, the solutions get stuck in a local optimum. Also the optimization starting from a worst-case solution is more easy than starting from solutions that are more optimal. In these cases, the five restarts from the worst-case solution performs better than the ten times iterative improvement of a single solution.

3.6 Conclusion and Further Research

In this chapter we have seen several approaches for solving the Blocking Job Shop Problem. Previous methods for the Standard Job Shop Scheduling Problem can not be applied effectively. Therefore new methods have been developed, that use different kind of approaches. Central in solving the Blocking Job Shop Problem is the existence of new types of infeasibilities. The addition of the blocking constraints make the Blocking Job Shop harder to solve.

The Cycle Analysis, presented in Section 3.3, gives a better insight on how these new infeasibilities occur. Being inspired by the simulation model of the standard Job Shop Scheduling Problem, a new objective function method called the Simulated Schedule Observer is developed, that can calculate the objective value in linear time. The SSO also gives us the ability to specifically identify why some solution are considered infeasible and at what point in time these infeasibilities occur.

The several Random Insertion solution approaches show how effective a larger neighbourhood is compared to classical approaches. Furthermore, the formulation used for the Random Insertion method can be extended to produce a new tool for solving existing infeasible situations. When focussing on future research, the application of emergency buffers can aid search processes to find new optima. Combining the results from partial feasible solutions, the landscape of solutions can be explored in a greater fashion.

In the process of applying emergency buffers, a decision needs to be made what operations should go into the emergency buffer. This decision, although looking straightforward, is however not obvious. Several operations can belong to multiple cycles. Also the resulting makespan depends on what operation is going to the buffer. The timestamp on which the operation is placed in the emergency buffer, determines the timestamp in which the other machines participating in the cycle can continue. More research needs to be conducted to gain a better insight in this problem.

Instance	Size	GK	JIH	OPT	RCI		RI		Extended RCI				Extended RI			
					Min	Avg	Min	Avg	Min	Initial	Pre-Opt	Post-Opt	Min	Initial	Pre-Opt	Post-Opt
abz5	10x10	1595	1782	1468	1570	1595,0	1560	1601,6	1578	6446	2129,2	1579,5	1468	6446	2308,1	1547,3
abz6	10x10	1222	1290	1145	1210	1232,2	1220	1244,6	1162	4526	1862,5	1169,3	1145	4526	1787,6	1190,1
abz7	20x15	1224	1103		1051	1068,4	1023	1062,4	1016	6122	1824,5	1054,5	986	6122	1916,5	1027,5
abz8	20x15	1226	1198		1016	1092,8	1051	1093,6	1046	6343	1937,3	1077,5	1014	6343	2018,6	1061,9
abz9	20x15	1166	1139		1036	1060,8	1047	1062,2	1003	6021	1865,1	1048,9	1030	6021	1961,5	1057,4
ft06	6x6	63	63	63	63	63,0	63	63,0	63	152	94,2	63,0	63	152	81,4	63,0
ft10	10x10	1103	1247	1068	1068	1091,2	1068	1112,6	1068	3394	1783,8	1091,1	1087	3394	1797,1	1096,4
ft20	20x20	1495	1566		1439	1456,0	1416	1443,0	1430	3220	2195,9	1456,7	1435	3220	2123,5	1449,6
orb01	10x10	1268	1322	1175	1175	1224,6	1238	1259,6	1215	2789	2082,1	1235,5	1175	2789	1939,5	1188,9
orb02	10x10	1092	1243	1041	1070	1117,4	1056	1077,0	1069	3869	1833,2	1082,6	1041	3869	1582,8	1059,6
orb03	10x10	1188	1256	1160	1163	1184,2	1164	1178,0	1160	2223	1803,2	1173,4	1163	2223	1836,2	1173,5
orb04	10x10	1203	1256	1146	1197	1221,2	1206	1231,0	1146	4038	1783,4	1172,8	1183	4038	1725,3	1197,4
orb05	10x10	1083	1088	995	1050	1084,6	1038	1057,0	995	3193	1241,8	1009,7	1029	3193	1482,8	1048,4
orb06	10x10	1265	1218	1199	1220	1249,2	1237	1246,2	1221	3644	2048,8	1243,5	1199	3644	1987,6	1228,8
orb07	10x10	517	570	483	494	507,6	506	521,2	492	1792	768,0	499,5	498	1792	734,5	502,9
orb08	10x10	1028	1113	995	1018	1034,4	1036	1043,4	1018	1919	1462,5	1023,5	1018	1919	1665,1	1033,4
orb09	10x10	1046	1192	1039	1089	1114,8	1062	1102,0	1046	3975	1451,3	1061,3	1048	3975	1459,9	1054,4
orb10	10x10	1153	1237	1146	1146	1172,4	1165	1194,6	1148	3906	1726,2	1161,5	1148	3906	1738,2	1158,7
la1	10x5	832	899	793	793	827,2	793	814,8	793	2272	1038,7	795,5	818	2272	1035,4	819,6
la2	10x5	793	878	793	815	833,4	814	829,2	815	1962	1196,0	817,5	815	1962	1110,3	823,5
la3	10x5	747	797	715	715	735,4	725	742,8	715	1588	863,4	718,2	721	1588	903,4	729,3
la4	10x5	769	814	743	756	768,0	743	759,8	743	2195	1138,1	757,9	756	2195	898,7	757,2
la5	10x5	698	745	664	671	692,2	671	691,8	671	1779	870,9	679,4	680	1779	1058,9	687,9
la6	15x5	1180	1275	1060	1135	1147,0	1105	1132,8	1114	2974	1371,3	1119,2	1060	2974	1544,0	1095,1
la7	15x5	1091	1122	1016	1053	1069,8	1042	1071,0	1033	2604	1443,2	1059,9	1040	2604	1410,4	1053,4
la8	15x5	1125	1342	1040	1088	1132,2	1074	1107,2	1071	2950	1554,2	1116,1	1090	2950	1584,2	1107,1
la9	15x5	1223	1387	1141	1224	1236,4	1204	1225,0	1202	3107	1443,5	1223,3	1212	3107	1435,3	1220,0
la10	15x5	1203	1284	1096	1123	1170,0	1165	1181,2	1124	3382	1767,8	1156,7	1141	3382	1846,9	1164,7
la11	20x5	1584	1622		1518	1548,8	1489	1525,8	1469	3891	1968,7	1515,3	1446	3891	2133,5	1466,5
la12	20x5	1391	1405		1300	1323,2	1320	1331,8	1298	3452	1746,0	1323,2	1295	3452	1795,1	1316,0
la13	20x5	1548	1801		1429	1469,6	1457	1498,2	1445	3795	2014,4	1482,1	1428	3795	2017,0	1454,3
la14	20x5	1620	1665		1532	1567,6	1468	1544,4	1502	4440	2153,2	1550,1	1487	4440	1965,5	1498,2
la15	20x5	1650	1601		1504	1547,6	1513	1545,6	1515	3949	2282,2	1541,8	1490	3949	2098,0	1514,3
la16	10x10	1109	1201	1060	1084	1110,4	1125	1135,2	1060	3898	1683,6	1083,7	1060	3898	1855,1	1086,8
la17	10x10	982	1098	929	934	956,4	934	976,4	930	3919	1461,3	944,5	930	3919	1525,4	931,5
la18	10x10	1078	1103	1025	1063	1078,8	1040	1063,4	1046	4513	1718,9	1061,5	1026	4513	1589,6	1068,5
la19	10x10	1093	1163	1043	1053	1103,6	1063	1093,0	1088	4465	1469,4	1098,5	1068	4465	1483,8	1079,3
la20	10x10	1118	1239	1060	1060	1108,8	1060	1096,2	1101	3905	1573,4	1114,3	1074	3905	1791,5	1098,5
la21	15x10	1545	1712		1459	1525,8	1460	1503,4	1464	6236	2554,9	1506,9	1398	6236	2258,9	1489,2
la22	15x10	1458	1393		1345	1389,6	1297	1385,4	1317	5976	2046,8	1372,4	1336	5976	2088,8	1374,8
la23	15x10	1611	1637		1454	1503,4	1477	1513,8	1483	6718	2194,8	1494,8	1418	6718	2422,0	1486,3
la24	15x10	1571	1695		1393	1434,8	1425	1448,6	1398	6494	2334,6	1432,2	1409	6494	2309,6	1437,8
la25	15x10	1499	1560		1356	1412,8	1421	1473,8	1375	6084	2288,1	1419,9	1354	6084	2193,9	1390,6
la26	20x10	2162	2421		1939	1964,6	1887	1965,8	1888	8608	3247,5	1948,6	1884	8608	2942,8	1923,0
la27	20x10	2175	2182		1959	1999,6	2019	2038,8	1944	8881	3245,4	2041,0	1973	8881	2994,9	2017,0
la28	20x10	2071	2201		1979	2027,6	1918	1995,2	1930	8241	2999,1	1990,9	1831	8241	3032,8	1924,7
la29	20x10	2124	1882		1808	1859,2	1771	1872,0	1793	7612	2955,1	1841,5	1751	7612	2875,5	1810,5
la30	20x10	2171	2140		1898	2017,2	1861	1946,2	1903	8456	3453,4	1966,5	1926	8456	3143,4	1978,6
la31	30x10	3167	2932		2659	2792,2	2740	2839,8	2707	12364	4292,5	2757,6	2693	12364	4520,8	2762,9
la32	30x10	3418	3406		3027	3068,0	2941	3028,6	2929	12467	4523,1	2989,5	2936	12467	5085,9	3010,9
la33	30x10	3131	3023		2684	2752,6	2718	2740,6	2676	11463	4179,2	2730,5	2646	11463	4033,9	2693,3
la34	30x10	3205	2985		2749	2825,6	2839	2861,6	2738	11032	4319,5	2821,8	2704	11032	4701,9	2799,4
la35	30x10	3311	3195		2888	2939,8	2831	2871,2	2760	11487	4241,3	2809,4	2713	11487	4453,8	2795,6
la36	15x15	1932	1957		1751	1794,8	1763	1808,8	1640	8969	3396,1	1758,5	1659	8969	3380,3	1734,2
la37	15x15	2053	1991		1891	1926,6	1886	1916,2	1887	10072	3189,4	1921,8	1827	10072	3407,6	1871,2
la38	15x15	1875	1889		1689	1730,0	1683	1712,4	1612	8873	2926,5	1676,7	1625	8873	3124,7	1695,6
la39	15x15	1950	2038		1749	1809,8	1706	1737,2	1715	9098	3325,7	1751,7	1658	9098	2944,7	1709,8
la40	15x15	1936	1971		1697	1749,8	1709	1765,6	1678	9591	3147,1	1774,0	1687	9591	3092,1	1719,6

Table 3.1: Results of the Random Insertion (RI), Random Critical Insertion (RCI), Extended Random Critical Insertion (Extended RCI) and Extended Random Insertion (Extended RI) solution approaches on the standard literature Blocking Job Shop Problems with swap. The column min indicates the best found answer. Columns $\overline{\text{Pre-Opt}}$ and $\overline{\text{Post-Opt}}$ indicate respectively the average pre-optimization and average post-optimization objective value. Our results are compared against known results from the literature. The GK column contains results from [8], the JIH column contains the results from [26].

Instance	Size	MP	JIH	OPT	RCI		RI		Extended RCI				Extended RI			
					Min	Avg	Min	Avg	Min	Initial	Pre-Opt	Post-Opt	Min	Initial	Pre-Opt	Post-Opt
abz5	10x10	1838	1818	1641	1656	1748,8	1705	1726,8	1666	6446	2619,3	1706,5	1663	6446	2574,6	1687,7
abz6	10x10	1419	1590	1249	1258	1296,6	1268	1297,8	1258	4526	1823,0	1356,6	1268	4526	1840,9	1274,3
abz7	20x15	2675	1312		1104	1113,4	1090	1109	1061	6122	2023,7	1089,8	1032	6122	1828,7	1070,4
abz8	20x15		1223		1104	1134	1106	1145,8	1071	6343	1973,7	1119,4	1068	6343	2009,3	1107,5
abz9	20x15	1976	1204		1118	1133	1111	1155,2	1073	6021	2025,6	1102,9	1070	6021	2053,4	1093,4
ft06	6x6	74	73	69	69	69,6	69	69	69	152	97,8	69,0	69	152	110,3	69,0
ft10	10x10	1447	1158	1158	1203	1261,4	1180	1204	1209	3394	1747,4	1216,7	1158	3394	2021,5	1201,7
ft20	20x20	1796	1628		1544	1592	1557	1572,6	1537	3220	2544,3	1571,5	1521	3220	2637,4	1540,5
orb01	10x10	1335	1407	1256	1256	1337,6	1269	1292,2	1288	2789	2085,3	1314,5	1256	2789	2097,5	1271,8
orb02	10x10	1370	1269	1144	1162	1192	1144	1173,2	1161	3869	1712,5	1187,5	1149	3869	1820,0	1170,3
orb03	10x10	1407	1426	1311	1311	1322	1311	1327,4	1311	2223	1832,6	1317,0	1311	2223	1792,9	1313,1
orb04	10x10	1460	1405	1246	1246	1293,8	1262	1280,4	1317	4038	1925,5	1336,5	1287	4038	1908,8	1296,8
orb05	10x10	1414	1449	1203	1247	1340,4	1214	1241,2	1264	3193	1528,4	1304,5	1253	3193	1378,2	1253,0
orb06	10x10	1448	1449	1266	1295	1318,2	1289	1304,6	1275	3644	2220,5	1319,5	1286	3644	2101,9	1298,7
orb07	10x10	565	633	527	531	562,2	527	532,2	543	1792	832,0	556,7	529	1792	855,8	539,5
orb08	10x10	1223	1324	1139	1139	1150,8	1139	1154,6	1139	1919	1648,5	1153,4	1139	1919	1607,8	1140,2
orb09	10x10	1329	1221	1130	1130	1192,6	1130	1172,6	1149	3975	1704,1	1175,6	1158	3975	1572,7	1181,3
orb10	10x10	1612	1567	1367	1406	1456,2	1367	1414	1445	3906	1681,0	1492,6	1419	3906	1643,3	1420,4
la1	10x5	1066	1050	881	921	983,8	881	944,2	881	2272	1365,6	918,2	881	2272	1396,3	938,7
la2	10x5	1077	969	900	921	935,4	900	908,4	902	1962	1266,5	908,4	911	1962	1272,1	915,9
la3	10x5	884	884	808	808	857,8	808	842,6	852	1588	1044,9	855,1	821	1588	1000,4	824,2
la4	10x5	881	965	859	861	873	859	865,8	861	2195	1430,7	871,4	861	2195	1494,2	861,2
la5	10x5	995	891	732	742	766	742	749,8	747	1779	1100,6	774,8	732	1779	1079,5	744,5
la6	15x5	1568	1317	1194	1236	1270	1232	1259,2	1210	2974	1625,3	1227,7	1227	2974	1637,0	1249,6
la7	15x5	1553	1327	1127	1178	1223,8	1151	1197,2	1153	2604	1531,1	1165,7	1153	2604	1560,6	1162,5
la8	15x5	1386	1335	1173	1249	1276,8	1213	1256,2	1173	2950	1816,5	1218,6	1203	2950	1751,5	1225,3
la9	15x5	1579	1410	1305	1306	1374,6	1337	1363,8	1352	3107	1897,4	1374,2	1319	3107	1808,9	1334,1
la10	15x5	1629	1371	1218	1255	1304,8	1272	1318,6	1224	3382	1920,2	1273,1	1247	3382	1738,4	1263,2
la11	20x5	2017	1780		1609	1682,4	1656	1681,4	1631	3891	2284,0	1676,5	1618	3891	2418,5	1643,2
la12	20x5	1800	1603		1459	1479,8	1441	1483,2	1400	3452	2276,0	1457,5	1418	3452	2311,3	1452,7
la13	20x5	2024	1693		1603	1655,8	1602	1657,2	1612	3795	2437,4	1639,5	1536	3795	2364,5	1574,3
la14	20x5	2228	1894		1673	1691,2	1691	1703,8	1613	4440	2383,2	1690,8	1633	4440	2427,8	1656,1
la15	20x5	2226	1789		1656	1720,8	1673	1695,6	1662	3949	2278,6	1693,4	1628	3949	2674,7	1675,1
la16	10x10	1231	1345	1148	1193	1223,2	1148	1193,6	1154	3898	1724,9	1218,4	1148	3898	1788,0	1157,8
la17	10x10	1146	1061	968	979	1021,2	968	1017,2	1008	3919	1676,9	1051,6	979	3919	1597,2	993,3
la18	10x10	1334	1263	1077	1120	1220,8	1077	1114,6	1119	4513	1710,3	1141,0	1077	4513	1762,1	1080,1
la19	10x10	1314	1268	1102	1170	1231,8	1113	1154,4	1124	4465	1597,2	1137,9	1113	4465	1830,5	1135,3
la20	10x10	1357	1238	1118	1170	1202	1118	1181,2	1164	3905	1841,6	1196,0	1118	3905	1924,1	1149,0
la21	15x10	2475	1737		1643	1692,6	1649	1669,4	1570	6236	2550,2	1623,2	1568	6236	2541,4	1594,5
la22	15x10	2430	1589		1428	1483	1450	1490,6	1406	5976	2181,1	1433,1	1423	5976	2251,0	1447,9
la23	15x10		1940		1587	1629,4	1573	1641,6	1580	6718	2706,0	1612,7	1538	6718	2560,8	1570,5
la24	15x10	2489	1752		1486	1547,8	1531	1572,4	1450	6494	2486,7	1590,4	1474	6494	2427,9	1514,2
la25	15x10	2001	1845		1553	1627,4	1485	1544,2	1476	6084	2458,4	1530,6	1465	6084	2387,9	1500,9
la26	20x10	3147	2270		2019	2105,6	2050	2081,8	2023	8608	3285,6	2070,3	1938	8608	3173,7	1967,1
la27	20x10	3505	2497		2172	2215,4	2141	2200,4	2088	8881	3434,3	2109,8	2140	8881	3802,7	2142,1
la28	20x10	4419	2293		2086	2145,4	2160	2204	2104	8241	3340,5	2154,2	2108	8241	3416,3	2125,6
la29	20x10	3298	2273		1950	1993,6	1970	1994,6	1899	7612	3093,2	1932,0	1899	7612	3052,9	1955,6
la30	20x10	3208	2317		2101	2164	2074	2156,8	2054	8456	3472,6	2093,5	1951	8456	3378,2	1978,3
la31	30x10		3157		2871	2946,2	2899	2978,8	2883	12364	4504,8	2979,5	2867	12364	4594,4	2923,3
la32	30x10	5710	3701		3162	3275,2	3135	3224,8	3122	12467	5161,3	3177,8	3149	12467	5302,0	3207,2
la33	30x10		3071		2869	2947,8	2878	2933,6	2870	11463	4631,9	2884,6	2840	11463	4345,8	2863,8
la34	30x10		3185		2993	3032,6	2875	2977,8	2950	11032	4858,9	2990,2	2958	11032	5003,7	2963,1
la35	30x10	5955	3548		3025	3058,8	3024	3078,6	2845	11487	4758,2	2924,4	2810	11487	4794,7	2861,3
la36	15x15	3209	2088		1806	1864,8	1761	1853,6	1757	8969	3438,5	1900,6	1809	8969	3344,8	1826,3
la37	15x15	3811	2255		1929	2053,6	1966	2022,6	1921	10072	3662,9	1989,4	1946	10072	3355,4	1969,2
la38	15x15		1982		1763	1828,6	1749	1809	1789	8873	3424,0	1816,0	1741	8873	3164,8	1791,4
la39	15x15	2765	1969		1775	1826	1792	1850,2	1747	9098	3388,2	1812,0	1796	9098	3169,2	1811,0
la40	15x15		2099		1760	1861	1817	1865,4	1794	9591	3433,9	1847,1	1834	9591	3395,9	1860,8

Table 3.2: Results of the Random Insertion (RI), Random Critical Insertion (RCI), Extended Random Critical Insertion (Extended RCI) and Extended Random Insertion (Extended RI) solution approaches on the standard literature Blocking Job Shop Problems without swap. The column min indicates the best found answer. Columns $\overline{\text{Pre-Opt}}$ and $\overline{\text{Post-Opt}}$ indicate respectively the average pre-optimization and average post-optimization objective value. Our results are compared against known results from the literature. The MP column contains results from [16], the column JIH contains results from [26].

Bibliography

[1] J. Admas, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[2] D. Applegate and W. Cook. A computational study of the job shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[3] J.E. Beasley. Or-library: Job shop scheduling. Website, November 2011. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/jobshopinfo.html`.

[4] V. Cerny. Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[5] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[7] H. Gröflin and A. Klinkert. Feasible insertions in job shop scheduling, short cycles and stable sets. *European Journal of Operations Research*, 177:763–785, 2007.

[8] H. Gröflin and A. Klinkert. A new neighbourhood and tabu search for the blocking job shop. *Discrete Applied Mathematics*, 157:3643–3655, 2009.

[9] Gurobi Optimization Inc. Gurobi optimizer reference manual, 2012.

[10] M.L. Kammer, J.A. Hoogeveen, and J.M. van den Akker. Identifying and exploiting commonalities for the job-shop scheduling problem. *Computers & Operations Research*, 38(11):1556 – 1561, 2011.

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

[12] A.M. Law. *Simulation Modeling and Analysis*. Mc Graw-Hill, 2007.

[13] S. Lawrence. *Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques*. Ph.d thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1984.

[14] J.K. Lenstra and A.H.G. Rinnooy Kan. Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics*, 4:Discrete Optimization I, 1979.

[15] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[16] A. Mascis and D. Pacciarelli. Job-shop scheduling with blocking and no-wait constraints. *European Journal of Operations Research*, 143:498–517, 2002.

[17] J.F. Muth and G.L. Thompson. *Industrial Scheduling*. Kluwer Academic Publishers, Dordrecht, 1963.

[18] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[19] E. Nowicki and C. Smutnicki. Advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8:145–159, 2005.

[20] M. Pinedo. *Scheduling Theory, Algorithms and Systems*. Prentice-Hall, 2002.

[21] S.E. Ramaswamy and S.B. Joshi. Deadlock-free schedules for automated manufacturing workstations. *IEEE Transactions on Robotics and Automation*, 12(3):391–400, 1996.

[22] B. Roy and B. Sussman. Les problèmes dordonnancement avec contraintes disjonctives. Technical Report 9, SEMA, Paris, 1964.

[23] H. Samarghandi and T.Y. ElMekkawy. Efficient meta-heuristics for the no-wait job shop problem. April 2011.

[24] R. Sedgewick. Permutation generation methods. *ACM Compututing Surveys*, 9(2):137–164, June 1977.

[25] C.H.M. van Blokland and H.C. Kampman. Structurizing the blocking job shop problem without buffers, handling deadlocks. Technical report, Department of Information and Computing Sciences, Utrecht University, 2011.

[26] J. van den Broek. *MIP-based Approaches for Complex Planning Problems*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[27] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992.