Developing an improved branching structure for constraint programming applied to the job shop problem

Arwin Sleutjes

Department of Information and Computing Sciences
Utrecht University
The Netherlands
June 27, 2024

Supervisors
Dr. J.A. Hoogeveen
Dr. J.M. van den Akker
R.J.J. Brouwer MSc

Abstract

This thesis focuses on the job shop problem where we minimize the makespan. This problem has been studied exhaustively and is proven to be NP-hard. Past research has explored both approximation methods and exact methods to solve this problem. Constraint programming techniques have shown potential and they present some interesting unexplored paths for research. In constraint programming there is a balance between the search heuristic used to explore the solution space by adding constraints and constraint propagation used to reduce the solution space by making it consistent with some number of constraints or to deduce additional constraints. Opportunities for constraint propagation have been thoroughly studied. However, strategies that combine multiple opportunities can be further explored. Besides that, this thesis focuses on new search heuristics based on the notion of the propagation amount. All new ideas have been implemented and experimentally tested on well known instances from the literature. As results, we present an alternative edge finding constraint propagation algorithm based on ideas from previous literature. Besides that, we show that graph constraint propagation can best be applied more often and more locally. Lastly, we develop a search heuristic based on the observed full propagation amount which outperforms the full propagation amount search heuristic from previous literature.

Acknowledgement

I would like to thank my supervisors for their support and guidance throughout the process of working on this thesis and I would like to thank family members and friends who became invaluable rubber ducks for debugging my code and my thoughts.

Contents

1 Introduction

We will start our introduction with an anecdotal example of the kind of problem that can be solved with the methods mentioned in this thesis. Imagine a facility suitable for a physical strength training. Such a facility has a number of stations where exercises can be performed. Each of these stations can be used by one person at a time. A training at such a facility usually take place in groups where the participants each have their own schedule of exercises that they want to do in a specific order. Additionally some participants might need more time for an exercise than others. In other words the duration of an exercise depends not solely on the exercise but also on the participant. If everyone would just do as they pleased, then it might take a long time before everyone is finished and the facility has to be reserved for an equally long time.

Imagine, for example, a situation where a group of 60 people all want to start with the same exercise for one minute. After that exercise one person continues with a second exercise for an hour and the remaining people are done. If everyone does their first exercise before the one person gets to do so, then the training will take two hours. If, on the other hand, the one person gets to go first, then the remaining people can do their exercise one-by-one whilst the one person does their second exercise. In that case the training will take only one hour and one minute.

This example illustrates the effectiveness of some global schedule that describes the order according to which the participants get to do each exercise. Finding the global schedule that results in the shortest possible reservation or training duration might seem trivial in our example. However, it is a complex problem to solve in general.

The problem. The example described above is more formally known as a job shop problem. The name **job shop problem** describes a class of scheduling problems (Graham et al., 1979). This thesis focuses on the most common variant where the makespan is minimized. Using Graham's notation this variant is denoted by $J||C_{max}$.

With a job shop problem we are given a set of jobs and a set of machines. Each job consists of a sequence of operations (a.k.a. activities or tasks) and each operation has to be processed on a given machine for some processing time. The order in which operations appear in the sequence of their job imposes precedence constraints (i.e. an operation can only start processing once its preceding operation has finished processing). Besides that, each machine can only process one operation at a time or it can be idle and operations can not be interrupted. The goal with a job shop problem is to find a schedule that satisfies these constraints. Additionally, with $J||C_{max}$, a solution is required to have a minimum makespan, where the makespan is simply the maximum completion time over all operations. We will use the term job shop problem to refer to this variant of the problem. Section 2 provides a more formal definition of the problem, the model, the constraints and the objective.

The relevance. The job shop problem has applications in actual production facilities that manufactures products. Solving it optimally can reduce operational costs of these job shops. However, even a simplified variant of the problem with only 2 machines ($J2||C_{max}$) has been shown to be NP-hard (Garey & Johnson, 1979).

Related work. Since the problem is NP-hard, we can either try to solve it approximately in polynomial time or we can try to solve it optimally in exponential time. In the literature both avenues have been explored. Approximation methods that have been applied include: heuristics such as priority dispatch rules by Haupt, 1989 and the shifting bottleneck approach by Adams et al., 1988; and local search using various neighbourhoods and meta-heuristics such as simulated annealing by Van Laarhoven et al., 1992 and tabu search by Nowicki and Smutnicki, 1996. Exact methods that have been applied include: integer linear programming models solved using branch and bound (Carlier & Pinson, 1989) or branch and cut (Applegate & Cook, 1991) and constraint programming approaches such as the one by Cheng and Smith, 1997. Section 3 provides a more extensive overview of any work related to our problem and the listed methods.

Our plan. This thesis will focus on the exact method constraint programming. Any constraint programming method generally consists of four parts: the model, a search heuristic, constraint propagation and a backtracking strategy (Baptiste et al., 2001). The search heuristic and backtracking strategy roughly correspond to a branching scheme. Combined with constraint propagation they form a branching structure. This thesis focuses on exploring new branching structures. Section 3 provides more context to these concepts.

Our motivation. Our focus is motivated by previous work that has shown that there is room for improvement with the branching structure. van der Sluis, 2022 showed that the size of search tree can be significantly reduced if a better search heuristic is used. He showed this by using a search heuristic that determines all consequences for each possible choice at a decision point. This strategy has one problem. Determining all those consequences at each decision point takes a lot of time. We note that this strategy could be improved upon by keeping track of a heuristic score for each choice that approximates those consequences. We will explore that idea in this thesis. Besides that, we note that there are many sources on specific procedures for constraint propagation but it is not often mentioned how those procedures can and should be combined. We will also work on determining a good structure that does combine these various procedures.

The remainder of this thesis. Section 2 will give a more formal definition of the problem. Section 3 will give an overview of relevant related work. Section 4 will describe the research questions and how we aim to answer them. Section 5 will describe some fundamental implementation details. Section 6 will discuss

our ideas in more detail accompanied by results of related experiments and some discussion. Lastly, Section 7 will summarize the conclusions that can be drawn from this thesis.

2 The job shop problem

This section will provide a more formal definition of the job shop problem and some other relevant concepts.

2.1 The input

As input we are given a set of n jobs J_j $(j = 1, \ldots, n)$ and a set of m machines M_i $(i = 1, \ldots, m)$. Additionally, each job J_j consists of a sequence of m operations $O_{i,j}$ $(i = 1, \ldots m)$. Lastly, each operation $O_{i,j}$ has an assigned machine $\mu_{i,j}$ and a processing time $p_{i,j}$. Note that $O_{i,j}$ refers to the i-th operation of job J_j. If that operation is assigned to machine M_i, then that is purely coincidental. In other words, $\mu_{i,j}$ is not necessarily equal to M_i. Besides that, the operations are assigned to machines such that each job has one operation assigned to each machine. However, the order of assigned machines is arbitrary for each job. So, $\mu_{i,j}$ is not necessarily equal to $\mu_{i,k}$. Lastly, the processing times are positive and discrete. However, there are no relations between the processing times of operations. Again, any similarities are purely coincidental. Table 1 provides the data for an example instance of the problem.

job j	operation i	machine $\mu_{i,j}$	processing time $p_{i,j}$
0	0	0	2
0	1	1	7
0	2	2	7
1	0	1	2
1	1	0	4
1	2	2	6
2	0	0	2
2	1	2	2
2	2	1	3

Table 1: Example instance data.

In addition to the above notation, we will use $O(J_j)$ to refer to the sequence of operations of job J_j. Respectively, we will use $O(M_i)$ to refer to the set of operations assigned to machine M_i.

Input format. Before we continue, we briefly describe how problem instances are commonly provided. Input files for the problem are generally formatted as one line containing the number of jobs n and the number of machines m followed by n lines (one for each job), each containing m tuples of a machine and processing time. Besides this most widely used standard, there also exists a different specification by Taillard, 1993. Figure 1 presents the example instance data as a text file in the standard format.

```
3 3
0 2 1 7 2 7
1 2 0 4 2 6
0 2 2 2 1 3
```

Figure 1: Example instance file.

2.2 The model

Given the above definitions we can now describe how the job shop problem can be modelled using variables, domains and constraints. We will do so whilst referring to the (in)equalities in Equations 1a through 1e on the next page.

Variables. The problem asks for a schedule that satisfies some constraints and minimizes the makespan. Such a solution is represented by the start time of each operation in the instance. Therefore the variables consist of a start time $S_{i,j}$ for each operation $O_{i,j}$. Since the problem does not allow for preemption (i.e. operations can not be interrupted), we can define the completion time $C_{i,j}$ of an operation as the start time plus the processing time, as represented in Equation 1a.

Domains. The start time for each operation must be greater than or equal to zero, represented by Equation 1b. Additionally, in some algorithms we define an upper bound ub on the makespan. In other words, the maximum completion time can not exceed the upper bound. This is represented by Equation 1c. These two inequalities represent the domains of our variable start times (since the completion times are defined in terms of the start times and processing times).

Constraints. Besides the domain constraints, we have two more main types of constraints. Within each job, the operations have to be processed in order. In other words, operation $O_{i,j}$ must complete before $O_{i+1,j}$ can start. These precedence relations are also referred to as conjunctive constraints, since the conjunction of all these precedence relations must be satisfied. They are modeled in Equation 1d. Additionally, we have that each machine can process at most one operation at a time. Therefore, we have, for each pair of operations $O_{i,j}$ and $O_{k,l}$ assigned to the same machine ($\mu_{i,j} = \mu_{k,l}$) a pair of precedence relations describing which one of them is processed before the other. Such a pair of precedence relations is referred to as a disjunctive constraint since a disjunction of each pair of relations must be satisfied. Moreover, only one of the disjuncts of each pair can be satisfied at the same time. Equation 1e models these constraints.

$$C_{i,j} = S_{i,j} + p_{i,j} \qquad \forall_{i\in\{1\ldots m\}} \quad \forall_{j\in\{1\ldots n\}} \tag{1a}$$

$$0 \leq S_{i,j} \qquad \forall_{i\in\{1\ldots m\}} \quad \forall_{j\in\{1\ldots n\}} \tag{1b}$$

$$C_{i,j} \leq ub \qquad \forall_{i\in\{1\ldots m\}} \quad \forall_{j\in\{1\ldots n\}} \tag{1c}$$

$$C_{i,j} \leq S_{i+1,j} \qquad \forall_{i\in\{1\ldots m-1\}} \quad \forall_{j\in\{1\ldots n\}} \tag{1d}$$

$$C_{i,j} \leq S_{k,l} \vee S_{i,j} \leq C_{k,l} \qquad \forall_{\{O_{i,j},O_{k,l}\}\in O(M_x)} \quad \forall_{x\in\{1,\ldots,m\}} \tag{1e}$$

Equations 1d and 1e present precedence relations as a comparison between a start time and a completion time. In this thesis and in some related work such a precedence relation $C_{i,j} \leq S_{k,l}$ is sometimes written as $O_{i,j} << O_{k,l}$.

2.3 The solutions

As mentioned, a solution to a problem instance consists of a start time for each operation.

Gantt charts. Generally solutions to scheduling problems like the job shop problem can be visualised using a Gantt Chart (Clark, 1923). In such a chart the vertical axis lists the machines and the horizontal axis represents time. See Figure 2 for a Gantt chart of a solution to the example instance corresponding to the solution presented in Table 2.

job j	operation i	start time $S_{i,j}$	completion time $C_{i,j} = S_{i,j} + p_{i,j}$
0	0	0	2
0	1	2	9
0	2	14	**21**
1	0	0	2
1	1	2	6
1	2	6	12
2	0	6	8
2	1	12	14
2	2	14	17

Table 2: Example solution to the example instance.
(C_{max} in bold)

Unnecessary idle time. Since we want to minimize the makespan, we want the last operation to complete as early as possible. Therefore, it has to start as soon as possible. The only things that prevent all operations from starting at time instant zero are the precedence relations from the conjunctive and disjunctive constraints. In other words, given a set of precedence relations, we can

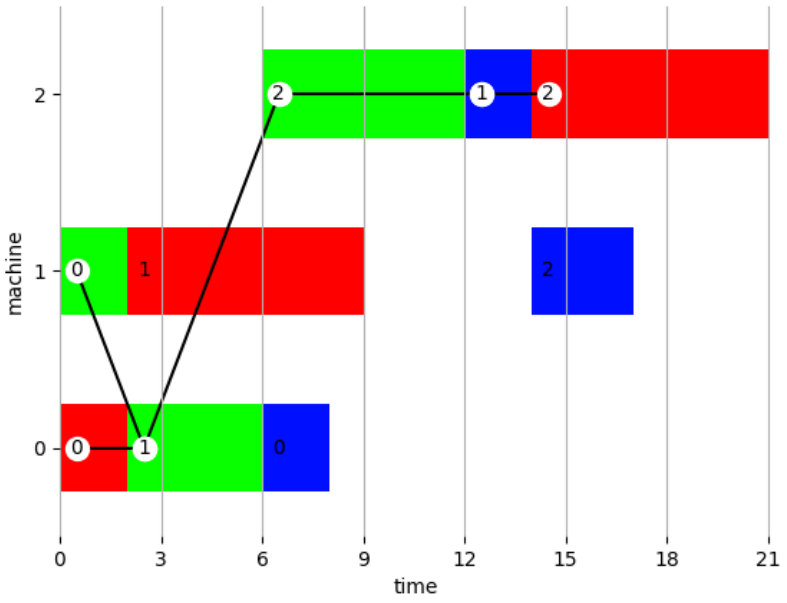

Figure 2: A Gantt Chart of the example solution.
($J_0 = red$, $J_1 = green$ and $J_2 = blue$)

find a solution with no unnecessary idle time by determining the corresponding earliest start times and the related makespan. This only works if we have decided or can deduce the direction of the precedence relation between each pair of operations within each machine (corresponding to the disjunctive constraints). Otherwise, two operations on the same machine could be scheduled to process at the same time. A schedule without unnecessary idle time is sometimes called an active schedule in the literature as described in Giffler and Thompson, 1960.

The precedence graph. The set of precedence relations can be modeled as a directed graph $G = (V, A)$, where we have a vertex $O_{i,j} \in V$ for each operation and a directed arc $(O_{i,j}, O_{k,l}) \in A$ for each precedence relation $C_{i,j} \leq S_{k,l}$. Such a directed graph is often referred to as a precedence graph. The precedence graph of the example instance including only the conjunctive constraints is presented in Figure 3.

Longest paths. If the precedence graph contains a cycle, then there is no feasible solution since two operations have to precede each other. On the other hand, if the precedence graph does not contain a cycle, then we can find the earliest start time for each operation by finding the longest path from any of the

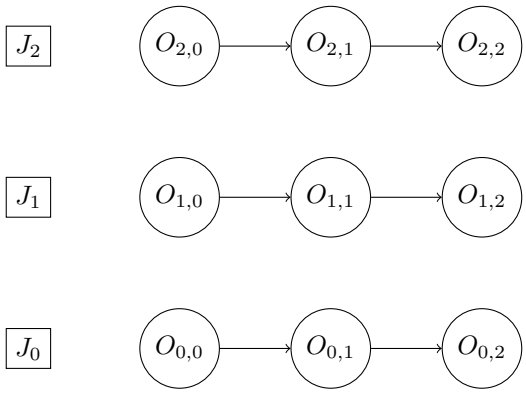

Figure 3: The precedence graph of the example instance including only the conjunctive constraints.

first operations of the jobs to that operation. The length of a path is defined as the sum of the processing times of the operations that are visited by the path. The completion time of each operation is then simply its processing time added to its earliest start time. In this case the maximum makespan can be found by taking the maximum over the completion times of the last operations of the jobs.

The process of finding the earliest start times and the makespan can be illustrated even better by adding two dummy nodes to the precedence graph. We add one source vertex s with zero processing time that precedes the first operations of the jobs. Respectively, we add one sink vertex t with zero processing time that is preceded by the last operation of each job. Now, finding the earliest start times is equivalent to finding the longest path from the source vertex s and the makespan is equal to the earliest start time of the sink vertex t.

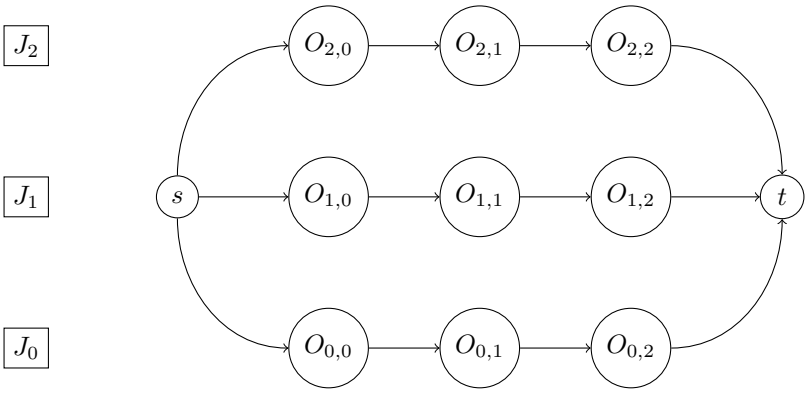

Figure 4: The precedence graph of the example instance including the conjunctive constraints and the source and sink.

The disjunctive graph. The precedence graph only represents the given conjunctive constraints for the jobs. It can be augmented with a set of undirected edges E representing the disjunctive constraints between pairs of operations assigned to the same machine. This augmented graph is referred to as a disjunctive graph $G = (V, A + E)$ in the literature (Roy & Sussmann, 1964).The disjunctive graph of our example instance is shown in Figure 5.

A disjunctive graph basically contains a clique of undirected edges on each set of vertices corresponding to the operations of one machine. Deciding or deducing the directions of the disjunctive constraints is equivalent to giving a direction to each of the undirected edges. Giving a direction to an edge is often called "selecting" and if we can deduce a direction (for example by transitivity), then we are speaking of an "immediate selection". If all edges have been selected, then the graph is called completely selected, otherwise it is partially selected. If the directed part of a partially or completely selected graph is acyclic, then it is called consistent. Figure 6 presents a consistent and completely selected version of the disjunctive graph of our example instance. It corresponds to the example solution which can be verified by determining longest paths as described.

For the interested reader, the concepts of a precedence graph and a disjunctive graph have been further generalized to the notion of a generalized temporal constraint network (Meiri, 1996).

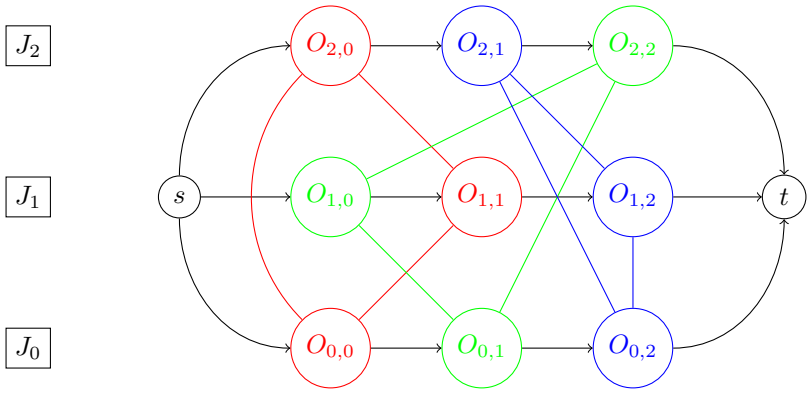

Figure 5: The disjunctive graph of our example instance.
($M_0 = red$, $M_1 = green$ and $M_2 = blue$)

Total enumeration. As we have seen, for every consistent and completely selected version of the disjunctive graph we can find a solution with no unnecessary idle time. Conversely, every solution (with or without unnecessary idle time) maps to a consistent and completely selected disjunctive graph. Lastly, any solution with unnecessary idle time can not be better than the corresponding solution with all that idle time removed. Therefore, we can exhaustively search the entire solution space by enumerating all consistent and completely selected disjunctive graphs and checking their makespan.

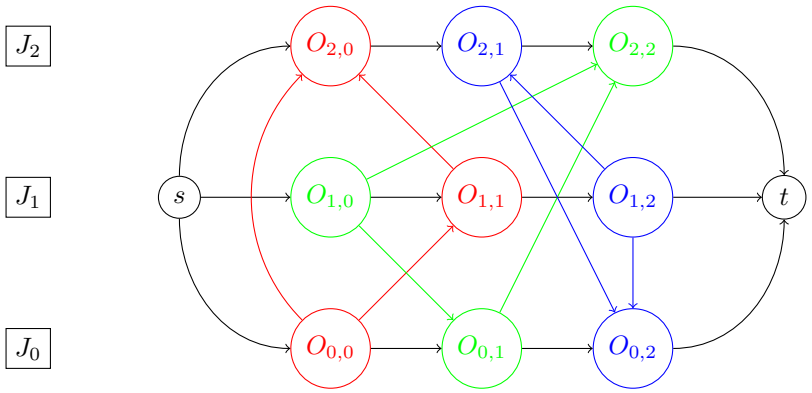

Figure 6: The completely selected disjunctive graph corresponding to our example solution.
($M_0 = red$, $M_1 = green$ and $M_2 = blue$)

Since there are n jobs, we have $\binom{n}{2} = \frac{n(n-1)}{2}$ edges per machine. With m machines that gives us $m * \frac{n(n-1)}{2}$ binary decisions. That means that there are $2^{m*\frac{n(n-1)}{2}} = O(2^{mn^2})$ different complete selections. Many of those complete selections might not be consistent but it does give an upper bound. Using topological ordering, determining the longest paths can be done in $|V| + |A| = |V| + |A| + |E| = (mn) + (m(n-1)) + (m * \frac{n(n-1)}{2}) = O(mn^2)$. These two bounds give us a total worst case time complexity of $O(2^{mn^2} * mn^2)$.

Intuition. Given the presented ideas and concepts we can develop some intuition regarding strategies to solve the problem. As we select disjunctive constraints one-by-one, we can keep applying a longest path algorithm to determine the earliest start time and makespan given the partial solution. If this partial solution has a makespan that exceeds some upper bound we can be sure that there is no feasible solution regardless of the choice we make for the remaining disjunctive constraints.

Machine sequences. Note that a complete selection of the edges of one machine can be represented by simple ordering of the operations by applying transitive reductions (if we have $(a, b), (b, c), (a, c) \in A$, then we can remove (a, c)). Conversely, a complete selection is the transitive closure of such an ordering. This gives a compact method to represent solutions as an ordering of operations for each machine. These orderings are referred to as machine sequences which leads us to the term machine sequencing that can be found in the literature. Figure 7 presents the example solution as a machine sequence where operations are represented by a tuple consisting of the job and the index within that job. Since each job has only one operation per machine this could be simplified further by dropping the index within the job.

0 0 1 1 2 0
1 0 0 1 2 2
1 2 2 1 0 2

Figure 7: Machine sequences of the example solution.

2.4 Additional definitions

In some cases we will refer to operations simply by O_i ($i = 1, \ldots n*m$). In those cases it should be clear from the context which operation this refers to. Furthermore, besides the processing time p_i and the assigned machine μ_i, there are some additional attributes of operations which will be used in some algorithms. We will define them here.

Operations can have a release date r_i and a deadline d_i. An operation can not start before its release date ($r_i \leq S_i$) and it can not end after its deadline ($C_i \leq d_i$). In some literature these terms are replaced by the earliest start time est_i and the the latest end time let_i. Given the processing time of each operation we can then also define the earliest end time $eet_i = est_i + p_i$ and the latest start time $lst_i = let_i - p_i$. All these terms refer to the window in which an operation can be scheduled (i.e. the domain of the variable start time of an operation).

Initially these attributes are determined by the sums of processing times of preceding and succeeding operations in the same job. These sums are sometimes called the head and tail respectively. As we select more disjunctive constraints these sums might increase resulting in tighter bounds on the domains of the variables. These bounds are induced by the selections that we make. Therefore, we speak of an induced release date, induced deadline, etc.

One important thing to note is the duality between release dates and deadlines. This gives rise to a symmetry that can be exploited to reuse algorithms in primal and dual form.

The concepts of release dates, deadlines, heads, tails, start times and end times can also be extended to apply to sets of operations by taking the minimum or maximum of those values among the operations in the set. It should be intuitive or clear from context whether such a term refers to the mininum or maximum.

3 Related work

This section will give an overview of relevant related work. There have been some excellent surveys in the past by Graham et al., 1979, Vaessens et al., 1996, Błazewicz et al., 1996 and Jain and Meeran, 1999. The last of those gives a very complete overview of the state of well known instances at that point in time. Da Col and Teppan, 2019 provide a good reference point of the recent state of the art of constraint programming solvers applied to the job shop problem. They applied OR-Tools (Google, 2024) and CP Optimizer (IBM, 2024) to the well known instances and experimented with some significantly larger instances. In Da Col and Teppan, 2022 they provide a more in depth analysis of the similarities and differences between those two solvers. We will not repeat too much from the mentioned works but we do want to touch upon the more important steps that have been made in the literature.

3.1 Instances

Over time, many different problem instances have been developed to test the performance of various algorithms for the job shop problem. We will go over important sets instances that by now form the classic set of benchmark instances.

Of the three instances (ft06, ft10 and ft20) by Muth and Thompson, 1963, the 10 x 10 (jobs x machines) instance (ft10) was one of the first surprisingly hard instances that sparked more interest in the job shop problem. It remained unsolved until 1986 (more than 20 years later) when it was solved using a method later reported in Carlier and Pinson, 1989. Later, Lawrence, 1984 generated 40 instances (la01-la40) of various sizes to accompany his PhD thesis. Applegate and Cook, 1991 published 10 more instances (orb01-orb10) generated based on input from visitors of a conference who were challenged to create hard problems. Storer et al., 1992 presented 20 more instances (swv01-swv20) with their article on new search spaces for the job shop problem. Yamada and Nakano, 1992 created 4 instances (yn01-yn04) to test their genetic algorithm. Taillard, 1993 generated a larger set of benchmark problems (ta01-ta80) of sizes representative of the real industrial problems at that time. Lastly, Demirkol et al., 1998 generated another set (dmu01-dmu80) with some larger instances and different routing structures through the machines.

van Hoorn, 2016 created a website with an overview of the mentioned instances accompanied by a table with lower bounds, upper bounds and references to the literature that presents those bounds.

3.2 Approximation methods

There are various methods to find an approximately optimal solution to an optimization problem. Broadly, there are two types of approximation methods. We have **heuristic methods** that iteratively **construct** a solution from scratch and there are **local search methods** that start out with an initial solution (usually obtained with a heuristic method) and iteratively **modify** it to improve

it. Both types of approximation methods can be designed in such a way that they always make locally optimal decisions with an increased risk of ending up in a local optimum as opposed to a global optimum. Methods that do this are often classified as greedy methods. This subsection will give an overview of approximation methods that have been applied to the problem.

3.2.1 Heuristics

As mentioned heuristic approximation methods construct a solution. Such a method is often based on some simple rule of thumb or intuition.

Priority rules (a.k.a. dispatching rules) are fast and intuitive heuristic methods used to solve scheduling problems. Haupt, 1989 provides a comprehensive survey of these methods. For some simpler scheduling problems, priority rules can quickly find an optimal solution. One example is the "earliest due date" (EDD) priority rule which optimally solves the one machine problem minimizing maximum tardiness ($1||T_{max}$) or lateness ($1||L_{max}$) as described in Jackson, 1955. Due dates are soft constraints similar to deadlines. If the completion time of an operation is greater than its due date then it is considered late or tardy. The lateness of an operation is defined as its completion time minus its due date. This can be negative. The tardiness is defined similarly, but it is bounded to be non-negative. For our problem, priority rules can only be used to find approximately optimal solutions.

Briefly, priority rules can be defined as policies that specify what to do in any situation where a machine becomes idle. Generally, you would want to schedule any available operation on such a machine. If there are multiple available operations competing for the same machine, then the priority rule is used to resolve that conflict. Priority rules make their decisions based on (parts of) the information in the problem instance.

A good example of a priority rule that can be described as a greedy approximation method for the job shop problem is the "shortest processing time" (SPT) priority rule. At any decision point, scheduling the operation with the shortest processing time will minimize the makespan of the next partial solution.

However, it will not be optimal in general. Take for example an instance with two machines. One job has its first operation assigned to machine 1 with processing time 2 followed by an operation assigned to machine 2 with processing time 100. Additionally there are 100 more jobs with their first and only operation assigned to machine 1 with processing time 1.

In this case SPT will schedule the operations of the 100 jobs on machine 1 before it schedules operation 1 of job 2 on that machine. This means that operation 2 of job 2 has to wait a long time before it can start processing resulting in a makespan of $100 * 1 + 1 * 2 + 1 * 100 = 202$. If we were to schedule operation 1 of job 2 first, then operation 2 of job 2 can start at time instant 2 and we would get a makespan of $1 * 2 + 1 * 100 = 102$.

A good example of a less greedy and very useful priority rule is "most work remaining" (MWKR). Here the intuition is that we pick the operation that has

the largest processing time including the processing time of its successors from the same job. Basically we want to process a part of the job that still has the most work to do as early as possible. This priority rule would optimally solve the problem instance described above because operation 1 of job 2 has more work remaining ($work = 2 + 100 = 102$) than the operations of the other 100 jobs ($work = 1$). The MWKR priority rule is an important part of other approximation and exact methods as will be mentioned in the following paragraphs and sections.

Note that some priority rules are more greedy than others. However the decisions made by priority rules are always final. Therefore priority rules should be considered greedy heuristics.

Even though we are focusing on an offline scheduling problem, it is interesting to note that priority rules also work for online variants of scheduling problems where the problem instance is given step by step since they can easily be adapted to decide based on only information from the past. The survey by Haupt, 1989 classifies common priority rules based on which information they depend on.

The survey by Błazewicz et al., 1996 places priority rules in a context of all existing methods to solve the job shop problem at that time. They note how, for a while, priority rules were the only methods that had any success with approximately solving problem instances of more than 100 operations. One effective method made use of probabilistic combinations of multiple priority rules as presented in a chapter of the book by Muth and Thompson, 1963. At each decision point this method would choose which priority rule to apply based on probabilities as opposed to always applying the same priority rule. Crowston et al., 1963 improved on this by combining priority rules into hybrids and applying a learning process to derive probabilities for the selection of priority rules.

The shifting bottleneck procedure is another important heuristic method that can approximately solve the job shop problem. Adams et al., 1988 noted, as we have mentioned in the previous paragraphs, that priority rules are rather greedy in the sense that all decisions are final. Therefore, the authors propose their shifting bottleneck procedure.

It works as follows. We have the set of all machines M and sequence them one by one. The set M_0 represents the set of sequenced machines. Initially no machines are sequenced. For each unsequenced machine $k \in M \setminus M_0$ we solve a one machine relaxation.

These relaxations are created as follows. We remove the disjunctive edges corresponding to all unsequenced machines $M \setminus M_0$ and replace the disjunctive edges corresponding to all sequenced machines M_0 by the conjunctive arcs as they have been sequenced.

Using a longest path algorithm from the source vertex, as we described in Section 2, we can now determine the release dates for the operations of the unsequenced machines and a makespan as induced by the graph. Note that the induced release date of an operation is simply equal to the amount of processing time required before an operation. This is also called the head as we described

in Section 2. Using the same longest path algorithm from the sink vertex in the same graph with all conjunctive arcs reversed we can also determine how much processing time is required after each operation as induced by the graph. As we mentioned in Section 2 this is also called the tail. Using the tails we can define a due date with respect to the determined makespan for all the operations of the unsequenced machines. We simply set the due date for an operation of an unsequenced machine equal to the derived makespan minus the tail of that operation.

Lastly, we can use the determined release dates and due dates to solve the one machine problem with release dates minimizing the maximum lateness $(1|r_i|L_{max})$. We will explain how these one machine problems can be solved at the end of this paragraph.

The unsequenced machine with the largest makespan given by the one machine relaxation is called the bottleneck machine. It is sequenced as in the solution to the one machine relaxation and added to the set of sequenced machines.

After sequencing a new machine, the set of sequenced machines is reoptimized. First, the sequenced machines are ordered by decreasing makespan. Next, those machines are resequenced one by one in that order. Resequencing is performed using the same one machine relaxation as before. The only difference is that the disjunctive arcs from the the machine that is being resequenced are also removed as opposed to replacing them by the correct conjunctive arcs. This cycle of ordering and resequencing is repeated three times after every newly sequenced machine.

After the last machine has been newly sequenced, the reoptimization cycle is repeated until no improvement has been made during a full cycle.

One important part of the shifting bottleneck procedure is the algorithm that solves the one-machine relaxations. Garey and Johnson, 1979 show that these relaxations are still NP-hard. McMahon and Florian, 1975 present an algorithm that can be used to solve these one-machine problems. Later, Carlier, 1982 developed a better method that can solve realistically sized instances in a matter of seconds with branch and bound by using a heuristic based on the MWKR priority rule (which we described before).

Vandevelde et al., 2005 performed a computational study on lower bounds for a generalization of this one machine problem with a variable number of parallel/identical machines.

3.2.2 Local search

As we mentioned before, local search methods start out with an initial solution (often obtained with a fast heuristic) and iteratively modify it to improve it. Such modifications are performed by considering a neighbourhood of solutions based on the current solution. Using that neighbourhood, a meta-heuristic is employed to decide which solution will replace (and hopefully improve on) the current solution. The neighbourhood and meta-heuristic are the main parts of any local search method.

The most basic meta-heuristic is a greedy one where we always move to a better neighbour as soon as we find one. It might be the case that there is an even better option in the neighbourhood but we pick the first one that is better than our current solution. This meta-heuristic is called "hill-climbing". Other common meta-heuristics are simulated annealing, variable depth search and tabu search (Glover & Laguna, 1997). Genetic algorithms are sometimes classified as local search algorithms as well.

We will briefly describe some interesting results from the literature. For more details, Vaessens et al., 1996 provide an exhaustive survey of the many forms of local search that have been applied to the job shop problem up to that point in time. Błazewicz et al., 1996 also present a nice overview.

Simulated annealing is a meta-heuristic for a local search procedure that is based on the idea of annealing hot metal. With simulated annealing we use a notion of temperature and an annealing schedule and we allow moves to worse neighbours in an attempt to not get stuck in a local optimum. The neighbour that will replace the current solution is chosen randomly, where the probability of choosing a certain neighbour depends in some way on its quality and the temperature. The chance that a worse neighbour is accepted is determined by the current temperature (a lower temperature lowers the chance of worse neighbours being accepted). The annealing schedule describes how the temperature changes and decreases over time.

Simulated annealing has a proof of convergence. This means that it will result in a global optimum, given enough time and the right annealing schedule. Granville et al., 1994 show that in practice the required time and annealing schedule often aren't feasible.

Simulated annealing has been applied to job shop scheduling by Van Laarhoven et al., 1992.

Genetic algorithms keep track of a population of solutions and apply operations that combine or modify them to (hopefully) improve the quality of the solutions in the population. The idea is that good characteristics of a number of solutions can be combined into one solution. Yamada and Nakano, 1992 apply a genetic algorithm to the job shop problem.

Tabu search is another meta-heuristic mechanism aimed at escaping local optima. With this method we keep track of a tabu list. Each iteration, once we make a move, the inverse of the move is appended to the tabu list. The tabu list has a maximum length and older moves are removed to make room for new moves. Whenever we are looking to make a new move, we are not allowed to make a move that is on the tabu list. Therefore, once we reach a local optimum and make a move to some neighbour, we can not immediately make a move back to that local optimum. Thus, we might escape from that local optimum. However, there is one exception. If a move on the tabu list leads to a solution

that is better than the current best known solution, then we allow it. Tabu search can be implemented as follows.

During each iteration, we split the neighbourhood in three sets. U (unforbidden), FP (forbidden and profitable/better than the current best known solution) and FN (forbidden and not profitable/worse than or equal to the current best known solution). Generally, we apply the best move from $U \cup FP$. If there is no such move (i.e. $|U \cup FP| = 0$), then we remove the oldest move from the tabu list and duplicate the youngest move at the front of the tabu list. If a move in the neighbourhood is now not forbidden, then we choose that one. Otherwise, we repeat the removal and duplication. In essence this means that we apply the oldest move on the tabu list that is also in our current neighbourhood and duplicate the youngest move on the list to keep the tabu list length the same.

Tabu search usually ends after a predefined number of iterations since the last time that we improved the current best known solution.

An expensive part of the tabu search method is the evaluation of the neighbouring solutions to determine which solutions are profitable. Taillard, 1994 aimed to alleviate this problem by applying some parallelization in their tabu search implementation for the job shop problem.

Nowicki and Smutnicki, 1996 achieved better performance by reducing the size of the neighbourhood that was used by the simulated annealing procedure of Van Laarhoven et al., 1992 and the tabu search approach by Taillard, 1994.

They also proposed a version of tabu search with backjump tracking. In this version, every time we improve on the current best known solution, we keep track of that solution, the current tabu list and the neighbourhood of that solution excluding the move we make in the next iteration and append those to a backjump list. Where we would usually end our tabu search after a number of iterations without improvement, we now go the the most recent element on the backjump list, restore that solution, tabu list and neighbourhood and we continue iterating. Additionally we remove the move we make in the next iteration from the neighbourhood in the backjump list. In essence this means that we keep track of a limited number of recent improvements to the global optimum and once we end a search we try alternatives from the neighbourhood of those improvements.

In their paper the authors also present some additional mechanism to reduce the runtime of the algorithm by detecting cycles in the search path and later, in Nowicki and Smutnicki, 2005, they present a further improved implementation of their algorithm.

Simulated annealing and tabu search have been combined by Zhang et al., 2008, hoping to exploit the good characteristics of both meta-heuristics in one algorithm.

Variable depth search aims to not get stuck in local optima by considering larger neighbourhoods. Since larger neighbourhoods become too expensive to

completely evaluate, variable depth search methods vary to which depth they explore a neighbourhood. This way, a larger part of the search space can be traversed and explored without computation becoming infeasible. Balas and Vazacopoulos, 1998 apply variable depth search to the job shop problem. Interestingly, in their implementation, the authors make use of the shifting bottleneck procedure, which we described previously, to guide their search.

3.3 Exact methods

In this section we will go over some important results from the literature regarding exact methods used to solve the job shop problem. The main approaches that have been used so far are general branch and bound, integer linear programming and constraint programming. Ideas from these approaches are often transferable from one to another with some slight modifications. We will go over some integer linear programming and constraint programming approaches.

3.3.1 Integer linear programming

Linear programming is a method that can be used to model decision and optimization problems. Linear programming starts by defining variables and linear constraints on these variables. A linear program can also have an optimization objective.

We will describe the idea in the context of a diet optimization problem for a hiking trip. In such a problem we have a few types of food that contain various ratios of macro nutrients like fat, carbohydrates and protein and we want to bring enough food to reach our daily goals for each macro nutrient whilst packing as little weight as possible.

The variables in this problem are the amount of each food type that we pack. As first constraints we have that we can not bring a negative amount of any food type. Secondly, we have constraints that require that we bring at least the daily goal's worth of each macro nutrient. Lastly, we might add constraints that represent a limited supply of any food type or we can consider how much volume they take up in our backpack. To complete the model we add an optimization objective that requires the minimization of the total weight of everything we bring. All these constraints can be expressed as linear (in)eqautions which makes it a linear program. Solutions to linear programs can be found using the simplex method developed by Dantzig et al., 1945.

Integer linear programming is an extension of the linear programming model that allows variables to be constrained to integral (and therefore also binary) values. Integer linear programs are not as straightforward to solve as linear programming problems. Integer linear programs are essentially combinatoric problems. Despite the similarities an (optimal) solution to an integer linear program might not be anywhere close to the optimal solution of the non integral version in the solution space.

The linear program example can be transformed by restricting the variables to integral values. The problem can then be changed from variables representing an amount of each food that is packed to an integral number of packages of each food type that is packed.

The model from Equations 1a through 1e can easily be repurposed to represent an integer linear program. In that case 1e can be implemented using two linear constraints from which one is selected to be active using a binary decision variable. Additionally, the upper bound (ub) from 1c becomes the objective variable that we want to minimize.

A naive method to find the optimal solution in an integer linear program is the total enumeration and evaluation of all possible combinations of values for the variables. In practice this is often infeasible. Therefore, some other strategies have been developed over time. These methods include but are not limited to branch and bound, branch and cut and branch and price.

Branch and bound is a procedure used to solve optimization problems. As the name suggests, it consists of two steps, branching and bounding, that are repeated to find a solution to a minimization (maximization) problem as described by Morrison et al., 2016.

Usually, before starting a branch and bound procedure, we find an initial feasible solution using some fast approximation method. The initial feasible solution is saved and its objective value is used as an initial upper (lower) bound for the minimization (maximization) problem. Alternatively, the initial upper (lower) bound is set to ∞ ($-\infty$).

Throughout the algorithm we explore a search tree with nodes representing parts of the solution space. We start at the root node representing the entire solution space. From there, we start branching and bounding.

We start with the branching step. From the current node we construct two (or more) child nodes by partitioning the part of the solution space represented by the current node using an additional constraint. The various methods of branching for a given problem are called branching schemes and the overall strategy describing which schemes to use is called the branching structure.

During the bounding step, for each child node, we calculate a lower (upper) bound on the objective value of solutions in the part of the solution space represented by that node. This is done by solving a relaxed version of the problem.

If the lower (upper) bound is greater (less) than or equal to the current upper (lower) bound, then we do not have to evaluate that part of the solution space any further and we can discard the corresponding child node.

If any of the child nodes are not discarded, then we continue the procedure by evaluating those child nodes. The order in which those child nodes are evaluated is described in the branching structure.

If, during the bounding step, the solution to the relaxed problem also satisfies all constraints of the unrelaxed problem (all values are integral in the linear relaxation), then we have found the optimal solution for that part of the solution

space. Next, we can update the upper (lower) bound with the objective value of that solution. If there are no more nodes left to be evaluated, then our search is complete.

Intuitively, better lower (upper) bounds will allow us to discard nodes earlier in the search tree. However, there is a trade-off because finding better bounds will generally require more time.

A lower bound for the makespan of the job shop problem can be found by solving a relaxed one-machine problem with release dates minimizing the maximum lateness ($1|r_i|L_{max}$) for each machine. As mentioned in our description of the shifting bottleneck procedure, McMahon and Florian, 1975 and Carlier, 1982 implemented algorithms to solve these relaxations.

Using that lower bound in a branch and bound algorithm for the job shop problem, Carlier and Pinson, 1989 are the first to solve the 10 x 10 instance (ft10) by Muth and Thompson, 1963. Each node in their search tree corresponds to a partial selection. The presented algorithm branches by choosing an unselected disjunctive constraint and creating a new node for both choices. Additionally the authors implement a method to derive immediate selections to reduce the size of the search tree. This method will later be known as edge finding.

In Carlier and Pinson, 1989, the authors implement edge finding in $O(n^4)$. Later, in Carlier and Pinson, 1990, they implement a faster edge finding algorithm that runs in $O(n^2)$. This faster variant is based on the preemptive variant of Jackson's schedule as described in Jackson, 1955. That is, the one-machine problem with release dates and preemption minimizing the maximum lateness ($1||r_i, pmtn|L_{max}$). In Carlier and Pinson, 1994 they reduce the time complexity to $O(n \log n)$ using a complex data-structure based on a binary search tree.

Branch and cut is very similar to branch and bound. It assumes that bounding is performed using the linear programming relaxation. After finding a solution to a relaxed problem, cutting planes can be added to tighten the relaxed problem. Cutting planes are additional constraints that are ideally violated by the relaxed solution but not by any feasible integral solutions.

Applegate and Cook, 1991 apply branch and cut to the job shop problem. The authors use cutting planes from some other authors and develop some themselves to find lower bounds. They present a branch and cut approach ("edge-finder") that uses ideas from Carlier and Pinson, 1989 with some enhancements. Additionally, they present a heuristic method that combines the shifting bottleneck procedure Adams et al., 1988 and the ideas from "edge-finder". That heuristic method is called "shuffle". Their final optimization strategy uses the shifting bottleneck procedure followed by an iteration of "shuffle" to find an initial upper bound. Afterwards, they use "edge-finder" to find and prove the optimal solution. This algorithm solves a large number of classic instances for the first time and is one of the first successful exact algorithms.

Branch and price can not always be applied to any problem because it requires a rather specific problem formulation. However, Martin and Shmoys, 1996 present an alternative formulation and algorithm for the job shop problem that could be considered a branch and price algorithm. The authors formulate the job shop problem as a packing problem for job-schedules. A job-schedule is an assignment of starting times to the operations of a single job such that the precedence constraints imposed by that job are satisfied. The goal is to find a job-schedule for each job such that no two jobs make use of the same machine at the same time, whilst minimizing the makespan.

3.3.2 Constraint programming

Constraint programming is a concept that can be compared to integer linear programming. It is aimed at solving combinatorial problems. In constraint programming we model problems as an instance of the constraint satisfaction problem and find a solution using a constraint programming system. The biggest difference between integer linear programming and constraint programming is that integer linear programs are solved using one clear algorithm whilst constraint programming is performed using multiple separate constraint propagation algorithms that operate on parts of the problem. Additionally, integer linear programs originate from the continuous mathematical optimization linear programs whilst constraint programming was designed from the ground up aimed at discrete combinatorial problems without a clear optimization objective.

The constraint satisfaction problem is a generalized model for combinatorial problems. Dechter, 2003 has written a great book on the constraint satisfaction problem. Informally, an instance of the constraint satisfaction problem can be defined as a triple of variables, domains and constraints.

The goal with a constraint satisfaction problem is to find an assignment of values to the variables from their respective domains such that all constraints are satisfied. This can be achieved with a constraint programming system as we will describe later. It is important to note that the constraint satisfaction problem does not allow for optimization objectives. A well known problem that can be modelled as a constraint satisfaction problem is a sudoku as shown in Figure 8.

The sudoku can be modeled as an instance of constraint satisfaction problem as follows. Firstly, we have a variable for each individual cell in the 9x9 grid. Secondly, we have as domain for each variable the numbers that are allowed in its respective cell (initially the integers 1-9). Lastly, we have as constraints the three main rules of a sudoku (no duplicate numbers within rows, no duplicate numbers within columns and no duplicate numbers within each of the nine 3x3 sub grids). The domains and constraints in the sudoku example are called explicit because they explicitly include, exclude or compare values for variables.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 8: A sudoku

Implicit domains and constraints can also be modeled in a constraint satisfaction problem. That way we can define domains as ranges of integral or real numbers and constraints as more complex inequalities. The domains of the variables in a sudoku can be modeled implicitly by bounding them to be greater than or equal to 1 and less than or equal to 9. Using such constraints we can model Equations 1a through 1e.

Constraint programming is a programming paradigm where the programmer declaratively specifies constraints over variables which a solution has to satisfy. Instances of the constraint satisfaction problem can be solved using constraint programming. Baptiste et al., 2001 state that the four main parts of a constraint programming system are: the model, a search heuristic, constraint propagation and a backtracking strategy.

The model describes how an instance of a problem will be formulated in terms of constraints and variables with domains. We previously described how to do this for a sudoku and we have already modeled the job shop problem in 1a through 1e. During constraint propagation we try to make the domains of the variables consistent with the constraints or we try to derive additional constraints. For sudoku this corresponds to trying to reduce the options for numbers that are allowed in a cell based on the constraints and the numbers that have already been filled in. At some point in a sudoku we might not be able to fill in additional numbers anymore without any doubt. In that case we have to make a guess using some intuition and see if it works out. If it does not work out we have to try some other idea. A situation that does not work out is usually called a contradiction. An example of a contradiction is a variable in a sudoku that does not have an assigned value yet but does not have any remaining values in its domain either. The intuition used to make a guess corresponds to the search heuristic in a constraint programming system and the

strategy that decides how to continue if our guess does not work out is called the backtracking strategy.

Constraint propagation rules are used to achieve consistency between domains and constraints or to derive additional constraints in a constraint satisfaction problem.

When we aim for consistency over all values in the domain of a variable we talk about D-consistency (referring to the domain). If we only aim to make the lower and upper bounds of the domains consistent with the constraint then it is called B-consistency (referring to the bounds). Ideally we would make the domains completely consistent with all the constraints at the same time. If we achieve that, then, for each variable and for each value within its remaining domain, we can find an assignment of values to all the other variables from their domains that results in a feasible solution. Such complete consistency is called n-consistency (n-D-consistency or n-B-consistency), where n refers to the total number of constraints. For completion we note that there is also a concept of k-consistency for some $0 < k < n$.

One problem with n-consistency is that it is generally not feasible to achieve efficiently. Therefore constraint programming systems use a limited but efficient set of constraint propagation rules to achieve some degree of consistency and a search heuristic and backtracking strategy to decide how to continue. For more information constraint propagation rules and consistency we refer to Błazewicz et al., 1996.

In our model, constraint propagation can be used to tighten the bounds of the start time of a variable, to make selections for some disjunctive constraints or to derive that no solution is possible given the current domains and constraints. We will discuss techniques to achieve this later in this section.

The constraint optimization problem is the name for the generalization of the constraint satisfaction problem that also includes optimization objectives. The book by Dechter, 2003 also describes this generalized problem. In the constraint optimization problem we want to find a feasible solution to the related constraint satisfaction problem whilst also optimizing a certain objective function. This search can be performed in many ways.

One interesting strategy to solve a constraint optimization problem is the following. Start with any feasible initial solution. This can be found with a fast heuristic or by solving the original constraint satisfaction problem. If we have any feasible solution, then we note that there might still be a better solution. However, now, we can restrict our search to a smaller range of objective values. The search can be restricted to this range by adding additional constraints. We already included these constraints in our model in Equation 1c. Let's say that the objective value of the last found solution is C_{max}. Then a modified constraint satisfaction problem can be constructed by taking the original constraint satisfaction problem and replacing the ub with $C_{max} - 1$.

If we find a new feasible solution by solving the modified constraint satisfaction problem, then we can repeat this procedure by constructing a new modified constraint satisfaction problem based on the new C_{max}. If, at some point, we can not find a solution in a modified constraint satisfaction problem, then we know that there is no solution with a better objective value than the last C_{max}.

Follow path is a strategy that can be used when solving an instance of the constraint optimization problem. The idea originates from the fact that the search heuristic in the constraint programming system made some choices to end up at the previous best solution. It might have taken some time to make these decisions and other good solutions might be nearby in the solution space. Therefore, once we modified the constraint satisfaction problem with a new upper bound, we follow the path of choices we made to end up at the previous best solution until we encounter a contradiction.

Constraint propagation rules for the job shop problem are summarized in Baptiste et al., 2001. We will briefly go over two of them that are often combined.

The disjunctive constraint propagation rule aims to derive immediate selections of individual disjunctive constraints and therefore tighter bounds on the start times. Disjunctive constraint propagation can be performed on any pair of operations on the same machine. The logic behind the rule is modeled in Equations 2a through 2d.

Equation 2a models the following idea. If we can not process two operations O_i and O_j, both assigned to the same machine, between the induced release date of O_i and the induced deadline of O_j, then O_j must be processed before O_i. The same holds when swapping O_i and O_j.

Equations 2b and 2c continue from there (in symmetrical fashion as mentioned in Section 2). If some operation O_i must precede some operation O_j, regardless of whether they are assigned to the same machine. Then the induced deadline of O_i must be less than or equal to the induced deadline of O_j minus the processing time of O_j. The same goes for the induced release date of O_j with respect to the processing time and induced release date of O_i.

Equation 2d models the fact that we encounter a contradiction when we can derive that two operations must precede each other. This becomes clear when you apply Equation 2a to both parts of the left-hand side of the implication.

$$\forall O_j \forall O_{i \neq j} [d_j - r_i < p_i + p_j] \implies [O_j << O_i] \tag{2a}$$

$$\forall O_j \forall O_{i \neq j} [O_i << O_j] \implies [d_i \leq d_j - p_j] \tag{2b}$$

$$\forall O_j \forall O_{i \neq j} [O_i << O_j] \implies [r_i + p_i \leq r_j] \tag{2c}$$

$$\forall O_j \forall O_{i \neq j} [d_i - r_j < p_j + p_i] \wedge [d_j - r_i < p_j + p_i] \implies \mathit{Contradiction} \tag{2d}$$

Equations 2a and 2d are most interesting to apply to pairs of operations that are part of a disjunctive constraint that has not yet been selected.

Equations 2b and 2c are only relevant to pairs of operations that have an established precedence relation.

The edge finding constraint propagation rule is in some sense a generalization of the disjunctive constraint propagation rule where we replace the operation O_j with a set of operations Ω. Edge finding is applied to the set of operations assigned to one individual machine. The logic of the edge finding constraint propagation rule is described in Equation 3, taken from Baptiste et al., 2001.

Before we explain the individual parts of this rule, we first want to explain some notation. Given a set of operations S, we have that r_S is the minimum induced release date over the operations in S, d_S is the maximum induced deadline over the operations in S and p_S is the sum of the processing times of the operations in S.

3a states that, if Ω and $O_i \notin \Omega$ (all assigned to the same machine) can not be processed between the induced release date of Ω and induced deadline of $\Omega \cup O_i$, then O_i must be processed before Ω.

3c continues from there. If O_i must be processed before Ω, then the induced deadline of O_i should be less than or equal to the minimum latest start time of any non-empty subset Ω' of Ω.

This might be somewhat hard to grasp, so we will elaborate a bit. Intuitively if $O_i << \Omega$, then we must have enough time after O_i to process the sum of the processing times of the operations in Ω before the latest induced deadline of the operations in Ω. This results in the bound on the induced deadline of O_i in Equation 3c. If we have $O_i << \Omega$, then we also have $O_i << \Omega'$ for any non-empty $\Omega' \subseteq \Omega$ and such a subset might result in a tighter bound on the induced deadline of O_i. Therefore, we take the minimum over the non-empty subsets of Ω.

Equations 3b and 3d present the same idea in the other direction. This is the symmetry that we touched upon in Section 2.

Equation 3e states that, if we have a set of operations that can not be processed between its release time and latest end time, then we have a contradiction. Notice that if its left hand side holds, then we can apply Equations 3a and 3b to derive two contradicting orderings.

$$\forall \Omega, \forall O_i \notin \Omega [d_{\Omega \cup O_i} - r_\Omega < p_\Omega + p_i] \implies [O_i << \Omega] \tag{3a}$$

$$\forall \Omega, \forall O_i \notin \Omega [d_\Omega - r_{\Omega \cup O_i} < p_\Omega + p_i] \implies [\Omega << O_i] \tag{3b}$$

$$\forall \Omega, \forall O_i \notin \Omega [O_i << \Omega] \implies [d_i \leq \min_{\emptyset \neq \Omega' \subseteq \Omega} (d_{\Omega'} - p_{\Omega'})] \tag{3c}$$

$$\forall \Omega, \forall O_i \notin \Omega [\Omega << O_i] \implies [\max_{\emptyset \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'}) \leq r_i] \tag{3d}$$

$$\forall \Omega [d_\Omega - r_\Omega < p_\Omega] \implies \mathit{Contradiction} \tag{3e}$$

Currently, the fastest algorithm for edge finding is that of Carlier and Pinson, 1994. We already touched upon this algorithm when we covered a lower bound for integer linear programming.

Baptiste et al., 2001 present another edge finding algorithm that runs in $O(n^2)$. Their algorithm is somewhat easier to adapt to other scheduling problems.

Note that both algorithms only update induced release dates and induced deadlines and we would need an additional procedure to explicitly make the immediate selections.

Search heuristics and backtracking schemes in a constraint programming system are very comparable to a branching scheme. They direct how we split up and search through our search space and what we do if we reach a dead end. If we can not derive any more information about the solution of a sudoku then the search heuristic is the part that decides how we continue. It can try to guess a value for a cell or it can split up the remaining possible values for a cell into multiple sets. If a search heuristic choice does not work out we try something else. The backtracking scheme decides how we do that.

The slack based search heuristic described by Smith and Cheng, 1993 has been applied in a constraint programming system for the job shop problem. The heuristic assumes that all possible immediate selections have been applied. Therefore, all remaining unselected disjunctive constraints have some slack in both directions such that Equation 2a can not be applied. The authors propose to make a decision for the disjunctive constraint that is the most constrained.

The formula for the slack of a disjunctive constraint on O_i and O_j in the direction $O_i << O_j$ is presented in Equation 4. Since we assume that all immediate selections have been made, the slack can not be negative. Otherwise, we should fix that disjunctive constraint in the opposite direction based on Equation 2a.

$$slack(O_i << O_j) = d_j - p_j - p_i - r_i \tag{4}$$

Now, when the authors need to make a decision, they find the disjunctive constraint with minimum slack in either direction and fix that constraint in the direction in which it has the most slack. Sometimes search heuristics for constraint programming systems are described in terms of orderings according to which we test variables and values for those variables. In this case we prioritize variables in order of increasing minimum slack in either direction and values in decreasing order of slack. Sadeh and Fox, 1996 explore some alternative orderings.

Smith and Cheng, 1993 also note that, if we use the described formula for slack to order variables, then variables are only ordered based on a part of the available information. The variable with the minimum slack in either direction is selected without any regard of the slack in the other direction and ties are broken randomly. The authors propose another search heuristic which does take the slack in the other direction into account. For a disjunctive constraint on O_i and O_j they define a measure of the similarity S of the slack values as shown in Equation 5.

$$S(O_i, O_j) = \frac{\min\{slack(O_i << O_j), \quad slack(O_j << O_i)\}}{\max\{slack(O_i << O_j), \quad slack(O_j << O_i)\}} \tag{5}$$

This measure of similarity is used to determine a biased variant of slack as shown in Equation 6. The authors propose to select the variable with the least biased slack in either direction. Values are always ordered the same regardless of whether we use the biased or unbiased slack since the denominator of biased slack will be the same for both values of each variable. With their new approach the authors still pick the value with the most slack.

$$biased\_slack(O_i << O_j) = \frac{slack(O_i << O_j)}{\sqrt{S(O_i, O_j)}} \tag{6}$$

In Cheng and Smith, 1997, the same authors also make use of their biased slack based search heuristic.

The impact based search heuristic as described by Refalo, 2004 is a general purpose search strategy for constraint programming. The idea is that we try to learn about the impact of decisions throughout our search. The authors start by describing a measure of the size of the search space. This can be used to define the impact of a decision in terms of the size of the search space before and after that decision. The authors define the size of the search space P as the product of the sizes of the domains of all variables. Given the size before P_{before} and after P_{after} a decision, they define the impact of that decision as shown in Equation 7. For this formula we have that $0 \leq impact(decision) \leq 1$ and if $impact(decision) = 1$, then we have reached a contradiction because there is a variable with an empty domain.

$$impact(decision) = 1 - \frac{P_{after}}{P_{before}} \tag{7}$$

For each variable we can determine the impact of assigning each of the values in its domain. Those impacts can be used to determine some measure of the impact of focusing our decisions or branching on a certain variable. The authors propose to do so by summing the impacts of the remaining values in the domain of a variable. Note that computing all these impacts at each node in a search before making a decision will take too much time. Therefore, the authors propose to define the impact as the average of the observed impacts of making a certain decision. This is possible because the same decision can be applied in different sub trees of our search tree.

During our search we can now focus on variables and values with maximum impact such that we hopefully reduce the search space as much as possible with each choice. Note that focusing on choices with maximum impact is different from guiding the search to a solution. Therefore, impact based strategies are more suitable when we have the idea that finding a solution will be hard, for example when a heuristic can not find it.

The authors also touch upon another problem with the proposed idea. If we only make use of observed impacts, then we don't have any information to base our initial choices on. However, these initial choices have a lot of influence on the rest of the search tree. Therefore the authors propose to initialize the impacts by testing many of the options at the start of our search tree before the first decision is made.

Lastly, the authors show that an impact based search heuristic outperforms purely focusing on variables with the smallest domain which is a strategy that can be compared to the slack based search heuristic. Additionally they show that initialization of impacts improves results.

The failure directed search heuristic as described by Vilím et al., 2015 can be compared to the impact based search heuristic mentioned above. The authors assume that the current solution is optimal and want to prove this with a small search tree. Failure directed search differs from the described impact based search heuristic in how choices are rated. Failure directed search puts more emphasis on making choices that fail immediately. The formula for the ratings in failure directed search is given in Equation 8. Note that R represents the reduction in effort necessary to explore the remaining search space and should have that $0 \leq R \leq 1$. One measure that can be used is $R = \frac{P_{after}}{P_{before}}$ defined in similar terms as the impact in the previous paragraph. Failure directed search prioritizes choices with lower rating (as opposed to higher impact in the impact based strategy) as they are hopefully more likely to result in contradictions quickly.

$$rating(decision) = \begin{cases} 0 & \text{if the branch fails immediately} \\ 1 + R & \text{otherwise} \end{cases} \tag{8}$$

The authors compare this rating strategy with the regular impact based strategy and show that failure directed search has better performance.

Additionally, the authors pay attention to the fact that ratings for the same decision will most likely be higher when they are made earlier (or higher up) in the search tree. Therefore, they keep track of the average rating of decisions at each depth of the search tree and use it to normalize ratings.

Lastly, the authors solve the problem of initializing the ratings by restarting the search after a number of backtracks.

The full propagation amount search heuristic described by van der Sluis, 2022 can be compared to the impact based search heuristic of Refalo, 2004. The author shows its potential in reducing the size of the search tree compared to the (unbiased) slack based search heuristic of Smith and Cheng, 1993. The full propagation amount search heuristic differs from the impact based strategy in how the impact is calculated. The full propagation strategy sums the absolute reduction of each domain whilst the impact based strategy calculates a ration that represents the reduction in size of the search space. We also note that

the full propagation amount strategy recomputes the impact of each possible choice at each decision point. As we mentioned during our explanation of the impact based strategy this is results in excessive run times. We will be working on applying the idea of only using observed scores with some initialization from the impact based strategy to the full propagation amount strategy.

3.4 The state of the art

Currently one of the best methods for solving scheduling problems like the job shop problem is large neighbourhood search coupled with a failure directed search as implemented in CP Optimizer (IBM, 2024).

Large neighbourhood search is comparable to local search. It starts with an initial solution. This is followed by a loop that aims to iteratively optimize this solution. The initial solution is relaxed, for example by replacing some conjunctive arcs by disjunctive edges again. For the relaxed problem a temporal linear relaxation is solved to find a lower bound as described by Laborie and Rogerie, 2016. Next, a completion strategy is applied to find a solution again. The relaxation strategy can be viewed as a local search neighbourhood and the completion strategy is comparable to the meta heuristic that decides where to move. CP Optimizer uses a portfolio of relaxation and completion strategies. Godard et al., 2005 describe the application of large neighbourhood search to a scheduling problem related to the job shop problem.

Failure directed search is added as a last resort after large neighbourhood search can not find any better solutions. It is used to quickly prove that there are no better solutions than the current best.

4 Research questions and methodology

In this section we enumerate the questions that we aimed to answer in this thesis and we will describe the methodology used to answer them.

4.1 Research questions

1. What is the effect of adding a follow path mode to the solver?
2. Can we develop a faster edge finding algorithm (w.r.t the $O(n^2)$ by Baptiste et al., 2001) without the complex binary search tree of Carlier and Pinson, 1994?
3. Can we improve the performance of graph propagation?
   (a) Can we limit the part of the graph that we propagate through?
   (b) Is there an alternative longest path algorithm that is more suitable for our application?
4. What is a good method for combining the various constraint propagation rules in a constraint programming system?
   (a) Does it help if we apply graph propagation immediately after applying edge finding or disjunctive constraint propagation to an individual machine?
   (b) Does it help if we only apply edge finding or disjunctive constraint propagation to machines where heads and tails changed since the last time we applied that propagation?
   (c) Does it help if we perform edge finding on machines in order of most critical to least critical?
5. Can we improve on the full propagation amount search heuristic by van der Sluis, 2022 to select a good choice to branch on?
   (a) Can we improve performance by using less costly propagation methods in our search heuristic?
   (b) Can we improve performance by using only observed full propagation amount values and some initialization similar to the impact based search heuristic by Refalo, 2004?
   (c) Can we improve performance by using a combination of the full propagation amount search heuristic by van der Sluis, 2022 and the observed full propagation amount strategy from the previous question?
6. How important is the quality of the initial upper bound?

4.2 Methodology

All the research questions correspond to variations in a constraint programming system aimed at solving the job shop problem. Such a system requires an initial upper bound.

We implemented a tabu search algorithm based on the algorithm described by Nowicki and Smutnicki, 1996 and a priority dispatch algorithm to find a starting point for that local search algorithm. Even though we implemented these algorithms, we ended up using the same problem instances and upper bounds as van der Sluis, 2022 to allow for a fairer comparison of performance of our results.

The instances are described in Table 3. The columns jobs and machines refer to the number of jobs and machines in the instances. The column opt contains the optimum makespan for each instance. These values are taken from van Hoorn, 2016. Lastly, the column init ub contains the initial upper bound from van der Sluis, 2022, which we will be using as well.

To answer our research questions we started with a basic constraint programming system which we will describe in the next section and we experimented with variations of this system to evaluate their effectiveness.

For question 1 through 4 each variant of the constraint programming system will also include the features introduced by the preceding questions.

For question 5 we will take the system that includes all features from question 1 through 4 and create three separate systems that only vary in their search heuristic.

For question 6 we compare the system from question 4 with the best result from question 5 using better upper bounds.

All algorithms have been implemented in Python 3.10.12. All experiment ran using the CPython interpreter on a personal desktop with an Intel Core i7-10700 CPU and 16 GB RAM running Ubuntu 22.04.3 LTS.

instance	jobs	machines	opt	init ub
abz5	10	10	1234	1276
abz6	10	10	943	976
ft06	6	6	55	55
ft10	10	10	930	1074
ft20	20	5	1165	1410
la01	10	5	666	666
la02	10	5	655	977
la03	10	5	597	653
la04	10	5	590	644
la05	10	5	593	593
la06	15	5	926	926
la07	15	5	890	985
la08	15	5	863	863
la09	15	5	951	951
la10	15	5	958	958
la11	20	5	1222	1222
la12	20	5	1039	1039
la13	20	5	1150	1150
la14	20	5	1292	1292
la15	20	5	1207	1251
la16	10	10	945	979
la17	10	10	784	795
la18	10	10	848	891
la19	10	10	842	893
la20	10	10	902	953
la30	20	10	1355	1536
la40	15	15	1222	1324
orb01	10	10	1059	1270
orb02	10	10	888	945
orb03	10	10	1005	1170
orb04	10	10	1005	1099
orb05	10	10	887	981
orb06	10	10	1010	1102
orb07	10	10	397	429
orb08	10	10	899	1048
orb09	10	10	934	1012
orb10	10	10	944	944

Table 3: Instance characteristics.

5 Implementation

In this section we will describe the implementation details of the algorithms that we used to answer the research questions. We start with a brief description of some important data structures and the strategy that we could use to find an initial upper bound, since that is required for a constraint optimization solver. As described in the methodology we ended up using the initial upper bounds from van der Sluis, 2022 instead of those found by our own implementation.

5.1 Important data structures

When an instance is parsed we create a tuple (job, job_index) for each operation where job corresponds to j and job_index corresponds to i for an operation $O_{i,j}$. The assigned machines and processing times are each stored in their own two dimensional array that is indexed using the tuples. Heads and tails are also each stored in their own two-dimensional array.

Machine sequences are implemented as a list containing one machine sequence for each machine. And each machine sequence is simply a list of tuples representing the operations as we described. This data-structure is mostly relevant for our tabu search algorithm. Each individual machine sequence can only contain at most one copy of each operation that is assigned to the corresponding machine. If all the operations are present in their corresponding machine sequence, then we have a complete solution. Otherwise we have a partial solution.

The directed part of the disjunctive graph is implemented using adjacency sets for fast adding and removing of arcs. We keep track of two versions of the directed part. One version contains the arcs in their normal direction which we refer to as the primal precedence graph. The other contains all arcs in their reversed direction which we refer to as the dual precedence graph. This is necessary because we need fast successor enumeration in both directions.

The undirected part of the disjunctive graph is kept track of using a set of edges for each machine.

5.2 Finding an upper bound

To find a good upper bound we implemented a version of the tabu search algorithm by Nowicki and Smutnicki, 1996. We describe this algorithm in Section 5.2.3. That algorithm does require an initial feasible solution from which it can start its local search procedure. To find such an initial feasible solution we initially implemented a very naive algorithm as presented in Section 5.2.1. Later we also implemented a priority dispatch algorithm based on the most work remaining rule as presented in Section 5.2.2.

5.2.1 Naive algorithm

Our naive algorithm starts with empty machine sequences. Next, we iterate over the jobs. For each job, we iterate over its operations and append each

operation to the sequence of its assigned machine. This can never create cycles. However, the quality of the solutions is not great.

5.2.2 Priority dispatch algorithm

The most work remaining priority dispatch algorithm works as follows.

Before the algorithm starts, we calculate the work remaining for each operation. For each operation O_i this is simply equal to the sum of processing times of the operations after O_i in the same job. This is also called the tail as we mentioned before. These tail values do not change throughout the algorithm.

Furthermore, the algorithm keeps track of the time at which each machine finishes processing the operation that it is currently processing (the machine release dates).

Lastly, the algorithm uses one heap to prioritize which operation to consider for scheduling. Each element on the heap is a triple containing a reference to an operation and the current release date and tail of that operation. The heap will prioritize triples with a smaller release date. If the release dates are equal, then the heap will prioritize the triple with the larger tail. If those are also equal, then we prioritize in triples based on the order in which they appear in the instance. Note that the release dates of elements on the heap will be updated throughout the algorithm such that this ordering essentially corresponds to prioritizing available operations with the largest tail (i.e. the most work remaining).

The algorithm starts by setting the machine release dates to zero and pushing the first operation of each job onto the heap with a release date equal to zero and the tail value that we just calculated before.

The algorithm proceeds by taking the first element from the heap, call it **current**.

If the release date of **current** is less than the release date of the machine of the operation of **current** (the machine is busy), then **current** is pushed back on the heap with an updated release date equal to the release date of that machine.

Otherwise (the machine is available), the operation of **current** is appended to the sequence of its assigned machine. Additionally, the release date of its assigned machine is updated to be equal to the completion time of the operation of **current**. Lastly, if the operation has a successor in the same job, then that successor is pushed onto the heap with as release date the completion time of the operation of **current**.

If, after these steps, the heap is empty, then we are done. Otherwise, we repeat these steps.

We will briefly explain how this algorithm correctly schedules available operations with the most work remaining. There are two cases where we can have multiple available operations contesting to be scheduled. Either the predecessors of these operations all finished at the same time or the machines that they are assigned to were busy whilst the predecessors of these operations finished processing.

In the first case the release dates of the contesting operations will be the same because they are initially equal to the completion time of their predecessor as we described. Therefore, the heap will prioritize the available operation with the largest tail.

In the second case the release dates of the contesting operations will be the same because they have all been updated to the release date of the machine as we described above. Therefore, the heap will prioritize the available operation with the largest tail.

5.2.3 Tabu search algorithm

The tabu search algorithm which we implemented consists of two main parts: a neighbourhood and a meta heuristic.

The neighbourhood we use is the same neighbourhood as in Nowicki and Smutnicki, 1996. The neighbourhood roughly consists of swapping operations at the borders of blocks in critical paths. We can find this neighbourhood as follows.

We start by finding all critical paths. As mentioned machine sequences simply represent a solution. For such a solution we can apply a longest path algorithm to find the start time of each operation and the makespan which is simply the start time of the sink t. The operation that enforces this makespan is called the critical operation. There might be multiple critical operations. Think of the simple case where there are two jobs, two machines, each operation has unit processing time and both jobs visit the machines in different orders. In this case the second operation of both job ends at time instant 2 and they both enforce the makespan to be equal to 2.

The longest path to a critical operation is a critical path. Since there might be multiple critical operations, there might also be multiple critical paths. There might even be multiple critical paths to the same critical operation as was also seen in Figure 2.

The critical paths can be represented by a directed graph in the dual direction. This graph contains an arc $O_{i,j} \leftarrow O_{k,l}$ if $O_{i,j}$ is a critical predecessor of $O_{k,l}$. This is the case if, after finding the longest paths, we have that $S_{i,j} + p_{i,j} = S_{k,l}$. We can find all critical paths during one run of a longest path algorithm as follows. Whenever we update the longest path to an operation we also keep track of the preceding operation that caused the update. Additionally, if another operation does not update the longest path to some operation but it does match the path length to that operation, then we also add it to the critical predecessors of that operation.

A block in a critical path is simply a sequence of consecutive operations that are assigned to the same machine. We can take two consecutive operations in a block and swap their position in the machine sequence of their assigned machine to obtain another feasible solution. The moves we described so far form the neighbourhood that was described by Van Laarhoven et al., 1992.

Nowicki and Smutnicki, 1996 noted that swapping operations in the middle of a block will not have any effect because it doesn't reduce the length of that critical path. Therefore they proposed to only consider moves where we swap the first or last pair of operations in a block of a critical path. This is also what we implemented.

The meta heuristic from Nowicki and Smutnicki, 1996 that we implemented is a form of tabu search. This method keeps track of a tabu list. Each iteration, once we make a move, the inverse of the move is appended to the tabu list. The tabu list has a maximum length ($maxt$) and older moves are removed to make room for new moves.

Additionally, during each iteration, we split the neighbourhood in three sets: U (unforbidden), FP (forbidden and profitable/better than best solution found so far) and FN (forbidden and not profitable/worse than or equal to the best solution found so far). This is done by first filtering out the unforbidden moves. For each forbidden move we evaluate the makespan of the associated neighbour to determine whether they are profitable or not.

Next, we take the best move from $U \cup FP$. If there is no such move ($|U \cup FP| = 0$), then we remove the oldest move from the tabu list and duplicate the youngest move at the front of the tabu list. If a move in the neighbourhood is now not forbidden we choose that one. Otherwise we repeat the removal and duplication. In essence this means that we apply the oldest move in the tabu list that is also in our current neighbourhood and duplicate the youngest move on the list to keep the tabu list length the same.

We keep iterating until we reach a predefined number of iterations ($maxiter$) since the last time that we improved the best solution found so far. At this point our basic variant of the tabu search algorithm returns the best solution found so far.

Nowicki and Smutnicki, 1996 also proposed an idea which they called back-jump tracking which we implemented as well. We did so as follows. In addition to the tabu list we created a backjump list with a maximum length $maxl$. This list is used as follows. If we improve the best solution found so far, then we append an element to the backjump list that contains that solution, the set of moves representing the neighbourhood of that solution and a copy of the tabu list at that point. We remove the move we make next from the set of moves in that element.

Now, instead of returning the best solution found so far after $maxiter$ iterations, if the backjump list is not empty, then we restore the solution and tabu list from the most recently added element on the backjump list and we choose our next move from the set of moves stored in that element. After making that move we remove it from the set in that element. If the set of moves is now empty we remove that element from the backjump list.

5.3 The basic solver

The basic solver is our initial constraint programming system which we adapt to test various algorithmic ideas to answer our research questions. For this system, we modelled the problem as in Equation 1. Furthermore, the basic solver was implemented as follows.

The constraint optimization system takes a complete description of a problem instance and an initial upper bound ub (which is based on a feasible solution found as described above). It contains one main loop that repeatedly finds a solution to the problem with an increasingly tighter upper bound. During one iteration of the main loop we call our constraint programming system to find a feasible solution with the instance description and the current upper bound. If that succeeds, then we save the solution and we update the upper bound to the makespan of the solution minus one. If it fails, then we return the most recently saved solution.

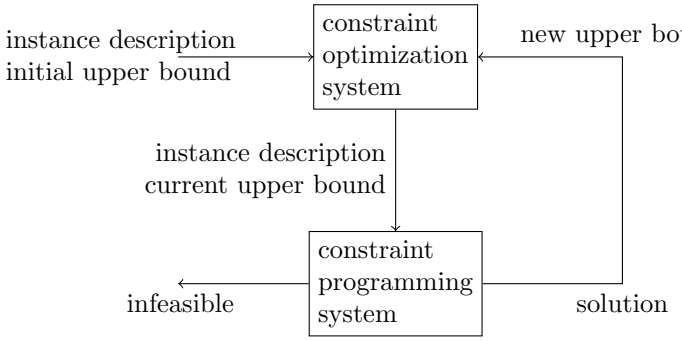

Figure 9: The constraint optimization system

The constraint programming system starts by constructing the initial disjunctive graph. The graph starts with a directed arc for each precedence relation from the conjunctive constraints on the jobs and an undirected edge for each disjunctive constraint. Next, we set the initial heads and tails to zero for all operations. Before, we go on, we immediately apply constraint propagation to make heads and tails consistent with the current disjunctive graph and to derive additional edges that can be fixed. This will initialize the heads and the tails correctly (equal to the sum of the processing times of preceding and succeeding operations respectively) and it will derive everything we can without making any heuristic guesses. If we encounter a contradiction at this stage we can never find a feasible solution and return.

Next, we initialize the branch path and the backtrack path as two empty lists. The branch path will be used to keep track of the branching decisions that have been made and whether there are unexplored alternatives for the decision. The backtrack path will contain for each branching decision a description of how

we can revert the changes that have been made to the disjunctive graph, heads and tails as a result of that branching decision and the following constraint propagation. Together these lists allow us to backtrack and continue our search when we encounter a contradiction. The backtrack list requires less memory than an implementation that saves explicit copies of the entire state (graph, heads and tails), however it might take a bit more time to revert changes.

Following the described setup, the constraint programming system enters a loop that continues as long as there are undirected edges in the disjunctive graph. The loop starts by selecting a branching decision (see below). Next, it applies that branching decision. Applying the branching decision consists of making the corresponding changes in the model and applying constraint propagation. If that succeeds, then we repeat the loop. If we encounter a contradiction, then we start backtracking. We will explain the underlined procedures in the following paragraphs.

If the loop ends that must mean that there are no more disjunctive edges and we have not encountered a contradiction. This means that we have a feasible solution (no cycles) that does not violate the upper bound. The system returns this solution to the solver.

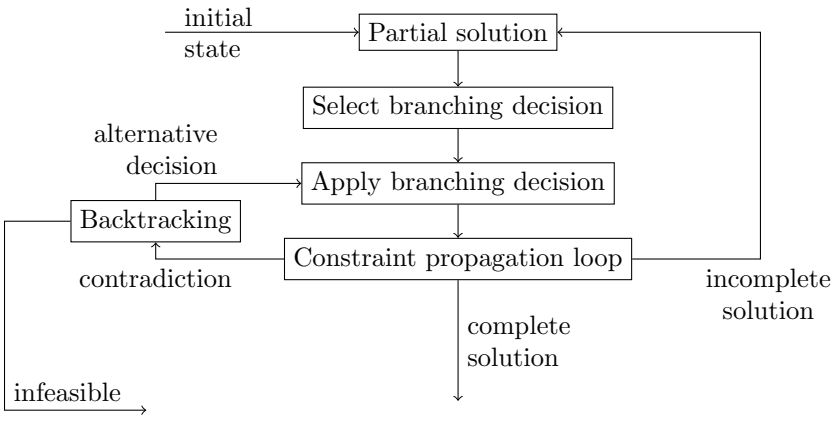

Figure 10: The constraint programming system

Selecting a branching decision is the procedure that corresponds to the search heuristic in the general definition of a constraint programming system. The basic solver implements a search heuristic based on slack as in Cheng and Smith, 1997. We describe it in Section 3.3.2. Basically, we find an arc with minimal slack corresponding to an unfixed edge and start by exploring the branch that fixes the edge in the opposite direction of that arc. As we mentioned we will adapt the basic solver with other search heuristics to answer our research questions.

Applying a branching decision consists of making changes corresponding to the branching decision and applying constraint propagation to derive tighter heads and tails and potentially more edges to fix.

For branching decisions where we fix an edge in some direction we start with modifying the disjunctive graph by removing the edge and adding the correct arc. Next, we apply Equations 2b and 2c to the new arc. These are described in Section 3. For a new arc $O_i \rightarrow O_j$ Equation 2b updates the induced deadline of O_i based on the induced deadline and processing time of O_j. Equation 2c updates the induced release date of O_j based on the induced release date and processing time of O_i.

Constraint propagation applies constraint propagation rules until there are no more updates from any of the rules. Applying one constraint propagation rule might create new opportunities for another rule. Therefore constraint propagation is implemented as a loop as shown in Figure 11. As mentioned in Section 3.3.2 there are a number of constraint propagation rules for the job shop problem. We have implemented the disjunctive constraint propagation rule and the edge finding constraint propagation rule. Additionally, we implemented a form of graph propagation which we further describe at the end of this section. We will explain how we implemented these three rules in the following paragraphs.

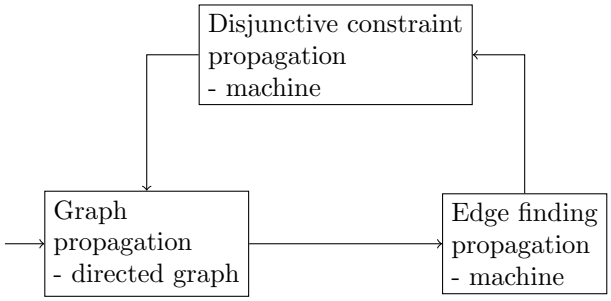

Figure 11: Constraint propagation loop

The disjunctive constraint propagation rule is applied to each machine individually. It applies Equation 2a to each unfixed edge in both directions to check if it can be fixed and immediately applies Equations 2b and 2c to update heads and tails along any new arcs.

The edge finding constraint propagation rule is applied to each machine individually as well. We described the general idea of this rule in Section 3. In our basic solver we implement the $O(n^2)$ edge finding algorithm by Baptiste et al., 2001. The pseudo code of that algorithm is presented in Algorithm 1. The pseudo code applies Equations 3b, 3d and 3e. A dual variant of the same algorithm applies Equations 3a, 3c and 3e.

We will explain how the pseudo code works for the operations of one machine x. The idea of the algorithm is to iterate over all relevant combinations of a set Ω and an operation $O_i \notin \Omega$. The algorithm requires that the operations of machine x are sorted and labeled (starting from 1) in order of increasing release dates. This can be done in $O(n \log n)$ time.

The algorithm fixes an operation O_k and defines Ω as the set of operations with an induced deadline less than or equal to that of O_k. Next the algorithm iterates over all operations and for each operation O_i that is not in Ω (based on the induced deadlines) it checks if we can process O_i and Ω between $r_{\Omega \cup O_i}$ and d_Ω (d_Ω is equal to d_k by definition). If we can not do so, then O_i must succeed Ω. This corresponds to Equation 3b.

The algorithm does not explicitly fix the edges between O_i and Ω. Instead it updates the induced release date of O_i to be at least as great as the maximum minimal completion time of any non-empty subset Ω' of Ω. This corresponds to 3d.

However, applying disjunctive constraint propagation after this algorithm will fix the correct edges. Carlier and Pinson, 1990 prove this.

In practice, the pseudo code in Algorithm 1 first determines the maximum minimal makespan of subsets that could be used to update release dates and then it checks if these updates should actually be applied. It does this once for each operation O_k. For a further and lower level analysis of the pseudo code see Appendix A

Graph propagation is used to propagate updates to heads and tails through the entire disjunctive graph and therefore between machines as opposed to the previous two rules that only update operations and edges of a single machine. As described before, the head (tail) of an operation is essentially the length of a longest path from (to) the start (end) of the disjunctive graph. Graph propagation should ensure that these longest paths are consistent with the current disjunctive graph and potentially updated heads (tails). In our basic solver we implemented this as finding a topological order in the primal (dual) directed part of the graph and updating heads (tails) for all operation in that order.

Backtracking is implemented as another loop. In that loop we take the most recent branching decision and the description of the resulting changes from the branch path and the backtrack path respectively. Using those, we first revert the described changes. After the changes are reverted, we check if there is an alternative to the branching decision from the branch path that we have not yet explored. If that is the case, we return that alternative branching decision to the constraint programming system. In that case the constraint programming system applies this alternative branching decision. If all alternatives have been explored for the last decision, then we repeat the backtracking loop for another iteration by taking the next elements from the branch path and backtrack path and repeat the above.

If there are no more elements on the branch path and backtrack path, then

that must mean that we have explored the entire solution space without finding a feasible solution that does not violate the current upper bound. In that case the constraint programming system is done and the last saved solution is optimal.

Algorithm 1 Edge finding by Baptiste et al., 2001

Require: That operations are sorted and labeled by increasing release date
Ensure: That Equations 3b, 3d and 3e are exhaustively applied

for $i \leftarrow 1$ up to n **do**
    $r'_i \leftarrow r_i$
end for
for $k \leftarrow 1$ up to n **do** ▷ Fix $d_\Omega = d_k$
    $P \leftarrow 0$, $C \leftarrow -\infty$, $H \leftarrow -\infty$
    for $i \leftarrow n$ down to 1 **do** ▷ Fix r_i
        if $d_i \leq d_k$ **then**
            $P \leftarrow P + p_i$
            $C \leftarrow \max(C, r_i + P)$
            if $C > d_k$ **then** ▷ Equation 3e
                Contradiction
            end if
        end if
        $C_i \leftarrow C$
    end for
    for $i \leftarrow 1$ up to n **do** ▷ Fix r_i
        if $d_i \leq d_k$ **then**
            $H \leftarrow max(H, r_i + P)$
            $P \leftarrow P - p_i$
        else
            if $r_i + P + p_i > d_k$ **then** ▷ Equation 3b and 3d
                $r'_i \leftarrow max(r'_i, C_i)$ ▷ Ω contains each O_j with $r_i \leq r_j \wedge d_j \leq d_k$
            end if
            if $H + p_i > d_k$ **then** ▷ Equation 3b and 3d
                $r'_i \leftarrow max(r'_i, C)$ ▷ Ω contains each O_j with $r_h \leq r_j \wedge d_j \leq d_k$
            end if ▷ Where O_h is the last operation that increased H
        end if ▷ Which means that $r_h < r_i$
    end for
end for
for $i \leftarrow 1$ up to n **do**
    $r_i \leftarrow r'_i$
end for

6 Results and discussion

The performance of the basic solver can be found in Table 4 under basic solver.

6.1 Adding a follow path mode

In this section we aim to answer the research question: What is the effect of adding a follow path mode to the solver?

This question originates from the fact that van der Sluis, 2022 does not provide a clear argument for why a follow path mode was added. To answer this we added a follow path mode to the basic solver. The results of the new solver can be found in Table 4 under follow path. By comparing it with the results of the basic solver we see that adding a follow path mode improves the performance of the algorithm.

- follow path performs better than basic solver for 30 instances.
  - Notably, follow path solves 1 instance that basic solver didn't solve.
- follow path performs equivalent to basic solver for 3 instances.
- follow path performs worse than basic solver for 4 instances.

This can be explained as follows. In any parts of the solution space where we previously found a contradiction we will definitely find a contradiction with a new upper bound. We shouldn't waste any time exploring those areas again to find that same contradiction. Therefore it works better to retrace the path to the previous best solution and continue our search from the first contradiction we encounter whilst doing so.

6.2 Improving edge finding constraint propagation

In this section we aim to answer the research question: Can we develop a faster edge finding algorithm (w.r.t the $O(n^2)$ by Baptiste et al., 2001) without the complex binary search tree of Carlier and Pinson, 1994?

This question originates from that fact that a number of papers in the literature van der Sluis, 2022 state that the $O(n \log n)$ algorithm by Carlier and Pinson, 1994 uses complex data structures and that many opt to use the $O(n^2)$ algorithm by Baptiste et al., 2001 instead.

To answer this question we implemented an edge finding algorithm based on that by Carlier and Pinson, 1994 excluding the binary search tree. The pseudo code for this algorithm is found in Algorithm 2. The algorithm builds Jackson's Preemptive Schedule and determines for each operation at its release date whether it is blocked by a set of operations. Jackson's Preemptive Schedule can be found with the most work remaining priority dispatch rule as we described before. We will briefly explain how we find a blocking set of operations.

Note that edge finding is performed on the operations of a single machine. For a given machine x this set of operations is denoted by $O(M_x)$. Throughout

the construction of Jackson's Preemptive Schedule on these operations, we keep track of the remaining processing time p_j^+ for each operation j. When an operation c is released we look for an operation s with some remaining processing time ($0 < p_s^+$) that satisfies Equation 9 (similar to Equation 5 from Carlier and Pinson, 1994) if any exists.

$$r_c + p_c + \sum_{\{j \in O(M_x) | 0 < p_j^+ \wedge q_s \leq q_j\}} p_j^+ + q_s > ub \tag{9}$$

Carlier and Pinson, 1994 use a binary search tree to search for such an operation s in $O(\log n)$ time. On the other hand, we simply iterate over the operations in order of decreasing tails and keep track of the sum of p_j^+'s in $O(n)$ time.

So, for each operation we use $O(n)$ time. This means that we still have total time complexity $O(n^2)$ for edge finding on a single machine as compared to the algorithm by Baptiste et al., 2001.

We took the solver that includes the follow path mode and replaced the edge finding algorithm by Baptiste et al., 2001 with the algorithm we just described. The results of this new solver can be found in Table 4 under edge finding. As can be seen, in practice, our algorithm has better performance.

- edge finding performs better than follow path for 32 instances.
  - Notably, edge finding solves 1 instance that follow path didn't solve.
- edge finding performs equivalent to follow path for 5 instances.
- edge finding performs worse than follow path for 0 instances.

It is also important to note that n is often rather small. Therefore, the performance gain of the algorithm by Carlier and Pinson, 1994 is also heavily influenced by their overhead (i.e. the constant factor that is omitted in time complexity analysis).

6.3 Improving graph constraint propagation

In this section we aim to answer the research question: Can we improve the performance of graph propagation?

This research question will be answered by answering two research questions in the following two subsections.

6.3.1 Partial graph constraint propagation

In this subsection we aim to answer the research question: Can we limit the part of the graph that we propagate through?

In the basic solver we apply graph propagation by finding a topological order over the entire graph and updating the heads and tails as longest paths in that order. Essentially we are recalculating the longest paths every time. We could

save some time here, since we already know the longest paths before the most recent changes were made.

Between two applications of graph propagation we can have two types of changes. If edge finding updates the head (tail) of an operation i, then only the heads (tails) of operations reachable in the primal (dual) directed part of the graph can be affected through graph propagation. The same goes for disjunctive propagation rule when we fix a new arc $O_i \rightarrow O_j$ and possibly update the head (tail) of O_j (O_i). To test the effect of this idea we keep track of all operations for which the head (tail) was updated since the last application of graph propagation and call them the sources. During the next application of graph propagation we construct a topological order that only includes those sources and the operations that they can reach in the current primal (dual) graph. This was added to the edge finding solver and the results can be found in Table 4 under partial graph. As we can see the partial graph propagation performs better.

- partial graph performs better than edge finding for 32 instances.
  - Notably, partial graph solves 2 instance that edge finding didn't solve.
- partial graph performs equivalent to edge finding for 5 instances.
- partial graph performs worse than edge finding for 0 instances.

6.3.2 Using an alternative longest path algorithm

In this subsection we aim to answer the research question: Is there an alternative longest path algorithm that is more suitable for our application?

So far we used a topological order to find and update heads and tails as longest paths. Alternatively we could use a kind of depth first algorithm to update the longest paths from the sources. In that case we can restrict further propagation to only those operations that were updated. That is, let i be one of the sources from which we are propagating heads. We only update heads of successors of i if the completion time of i is greater than the head of that successor. Next, we only continue propagation for successors whose heads we updated based on the completion time of i.

Theoretically there is no clear winner between these two alternatives, because there are no clear bounds or characteristics for the parts of the graph that are affected. The advantage of the topological order method is that, if there are multiple paths that can update the head of a certain operation i, then we propagate through all those paths before we propagate further from i. With depth first search we might propagate from i multiple times (roughly once for each path that updates i). On the other hand the depth first method can stop further propagation from successors as soon as they are not updated where the topological order includes all successors regardless of whether they are affected. We implemented a depth first variant of graph propagation and added it to the previous partial graph solver as replacement of the topological order graph propagation. The results of this new solver can be found in Table 4 under depth first.

- depth first performs better than partial graph for 30 instances.
- depth first performs equivalent to partial graph for 7 instances.
- depth first performs worse than partial graph for 0 instances.

6.4 Combining constraint propagation algorithms

In this section we aim to answer the research question: What is a good method for combining the various constraint propagation rules in a constraint programming system?

This question will be answered by answering three questions in the following three subsections.

6.4.1 Granular graph constraint propagation

In this subsection we aim to answer the research question: Does it help if we apply graph propagation immediately after applying edge finding or disjunctive constraint propagation to an individual machine?

Edge finding and disjunctive constraint programming can derive new constraints based on current heads and tails. If these heads and tails are tighter we might be able to derive more new constraints. Therefore, it might be beneficial to propagate updates to heads and tails immediately after an application of either of those rules before we apply that rule to the next machine as opposed to propagating once after applying all rules to all machines. This way any changes can be taken into account immediately. We add this idea to the depth first solver and present the results in Table 4 under granular graph.

- granular graph performs better than depth first for 32 instances.
- granular graph performs equivalent to depth first for 5 instances.
- granular graph performs worse than depth first for 0 instances.

6.4.2 Using flags to signal constraint propagation opportunities

In this subsection we aim to answer the research question: Does it help if we only apply edge finding or disjunctive constraint propagation to machines where heads and tails changed since the last time we applied that propagation?

As mentioned, edge finding and disjunctive constraint propagation derive new constraints based on current heads and tails. If, for a specific machine, these haven't changed since the last time we applied one of those propagation rules, then that rule will not derive any new constraints. Therefore we should not waste time on applying that rule to that machine again. We implement this by maintaining a flag for each machine for both rules. The flag for edge finding on a certain machine is turned on when disjunctive constraint or graph propagation update a head or tail on that machine and vice versa for disjunctive constraint propagation on a machine when edge finding or graph propagation

update a head of tail on a machine. We added this idea to the granular graph solver and the results of this solver can be found in Table 4 under flags.

- flags performs better than granular graph for 31 instances.
- flags performs equivalent to granular graph for 6 instances.
- flags performs worse than granular graph for 0 instances.

6.4.3 Prioritizing critical machines during constraint propagation

In this subsection we aim to answer the research question: Does it help if we perform edge finding on machines in order of most critical to least critical?

Intuitively the most critical machine might be more likely to present a contradiction or new constraints through propagation. All mentioned methods of edge finding can return the makespan of Jackson's Preemptive Schedule for the machine to which it is applied. For the $O(n^2)$ method from Baptiste et al., 2001 one can return the maximum value that C attains over all O_k. For methods that explicitly build Jackson's Preemptive Schedule one can simply return the makespan. That makespan is indication of how critical a certain machine is. We use the makespan determined during the most recent application of edge finding for each machine to sort the machines in order of most to least critical before we apply edge finding or disjunctive constraint propagation. This idea is added to the flags solver and the results of this solver can be found in 4 under critical.

- critical performs better than flags for 13 instances.
- critical performs equivalent to flags for 12 instances.
- critical performs worse than flags for 12 instances.

The results of flags and critical are roughly equivalent. One possible explanation for this is that the cost of sorting the machines each time outweighs the benefit of finding a contradiction or some new edges to fix earlier.

6.5 Improving the full propagation amount search heuristic

In this section we aim to answer the research question: Can we improve on the full propagation amount search heuristic by van der Sluis, 2022 to select a good choice to branch on?

To answer this research question we test three ideas to answer three questions in the following subsections. Before we do so, we start by implementing the full propagation amount search heuristic as described by van der Sluis, 2022. This will be used as a baseline to compare our ideas against.

As we described in Section 5, we have a procedure that applies a branching decision and we have the constraint propagation loop. Additionally we have a

backtracking procedure that can revert changes made by those two parts. While we apply a branching decision and the constraint propagation loop, we can keep track of the number of edges that we fix and by what amount we increase the heads and tails of the operations.

During the application of a branching decision and the constraint propagation loop, let r_i^δ be the amount by which we have increased the head of O_i. Respectively, let q_i^δ be the amount by which we have increased the tail of O_i. We can sum these amounts over all operations and add the number of new edges fixed ($edges\_fixed^\delta$). This is what we call the propagation amount (PA). It is presented in Equation 10. Note that the number of fixed edges contributes relatively little to this score. It could even be left out since the effect of fixing these edges also included in the form of increased heads and tails of other operations. We implemented it like this to keep our heuristic similar to the one by van der Sluis, 2022.

$$PA = edges\_fixed^\delta + \sum_{i \in \{1 \ldots n*m\}} r_i^\delta + q_i^\delta \tag{10}$$

The full propagation amount search heuristic works as follows. Whenever we need to select a new branching decision, we have to choose an edge to fix and an direction to fix it in. For each edge and direction we can simply apply that decision and the constraint propagation loop to calculate PA. Next, we can revert the changes made using our backtracking procedure. This way we can calculate the amount of propagation that each possible decision would give if it were applied at the current point in our search.

Let $PA_{i \to j}$ refer to the propagation amount resulting from fixing the edge between O_i and O_j to the arc $O_i \to O_j$. If we were to simply pick the edge and direction that has the largest $PA_{i \to j}$ and we eventually have to backtrack, then it might be the case that the alternative decision has a much smaller $PA_{i \leftarrow j}$. Therefore, we select the edge with the largest sum $PA_{i,j} = PA_{i \leftarrow j} + PA_{i \to j} (= PA_{j,i})$ and we fix it in the direction that has the largest propagation amount.

Note that calculating PA might result in a contradiction. If this happens for both directions of an edge, then we have reached a contradiction. There is an edge that can not be fixed in either direction without reaching a contradiction. Therefore, we have to start backtracking.

It might also be the case that only one of the directions for an edge results in a contradiction whilst calculating PA. In that case we can simply fix it in the other direction.

Lastly, the search heuristic by van der Sluis, 2022 only considers branching decisions for the edges on the most critical machine that has any remaining unfixed edges. We implemented this as well. We already explained how we can find the most critical machine in Section 6.4.3. We focus on the most critical machine that has any remaining unfixed edges.

We took the solver corresponding to the critical column in Table 4 and replaced the search heuristic with the described full propagation amount search heuristic to create a new solver. The results of the new solver can be found in

Table 5 under fpa.

Note, that fpa performs worse than critical on nearly all 37 instances. Therefore, this heuristic should definitely be improved before it can be used.

6.5.1 Using the partial propagation amount

In this subsection we aim to answer the research question: Can we improve performance by using less costly propagation methods in our search heuristic?

In the full propagation amount search heuristic we determine the propagation amount by applying a branching decision and constraint propagation as if we were normally branching. This way we know exactly what we can expect as a result of each option. However it takes a lot of time to determine the propagation amount for each choice.

Therefore, we implemented a variant of this search heuristic that uses less complex constraint propagation when we are calculating each PA. This way we can get an indication of the propagation amount that would result from that choice with less computation time. When we actually select a branch to apply we still fully propagate as before.

Less complex constraint propagation simply means that we do not use the edge finding constraint propagation rule during constraint propagation. Obviously, there are other methods for reducing the complexity of constraint propagation.

We call our new search heuristic: partial propagation amount. Similar to the full propagation amount search heuristic we focus on the most critical machine that has any remaining unfixed edges. Within that machine we select the edge that has the largest $PA_{i,j}$ value and we explore the direction that has the largest partial propagation amount first.

To test this solver we took the full propagation amount (fpa) solver and simply replaced the search heuristic. The results of the new solver can be found in Table 5 under ppa.

- ppa performs better than fpa for 21 instances.
- ppa performs equivalent to fpa for 4 instances.
- ppa performs worse than fpa for 12 instances.
  - Notably, ppa didn't solve 1 instance that fpa did solve.

6.5.2 Using the observed full propagation amount

In this subsection we aim to answer the research question: Can we improve performance by using only observed full propagation amount values and some initialization similar to the impact based search heuristic by Refalo, 2004?

This idea is inspired by the impact based search heuristic described by Refalo, 2004. It is based on the idea that the same branching decision might have similar results regardless of where we apply it in our search tree. Additionally, it uses the fact that we can keep track of the observed full propagation amount

resulting from each past application of a branching decision. Using these observed propagation amounts we can approximate the propagation amount that the same branching decision will have in an unexplored part of the search tree.

This is exactly what we do in our observed full propagation amount search heuristic. $OPA_{i \to j}$ will represent the expected full propagation amount of fixing $O_i \to O_j$ based on past observations. Whenever we actually select an edge and direction to fix it in $(O_i \to O_j)$, we update our measure as shown in Equation 11.

$$OPA_{i \to j} = 0.9 * OPA_{i \to j} + 0.1 * PA_{i \to j} \tag{11}$$

Note that we do need to initialize $OPA_{i \to j}$. At the start of our search we simple set $OPA_{i \to j} = PA_{i \to j}$.

The important difference between this this search heuristic and the full propagation amount search heuristic is the following. The observed full propagation amount search heuristic only uses the observed $PA_{i \to j}$ value when we actually apply a decision as part of our search. That observed value is then used to update $OPA_{i \to j}$. When we want to select a decision to apply we only use the $OPA_{i \to j}$ and $OPA_{i,j}$ values as they have already been calculated and updated. On the other hand, the full propagation amount search heuristic, calculates $PA_{i \to j}$ for each possible decision every time we need to select a new decision to apply in our search. The latter obviously takes a lot more time.

Just like our previous search heuristics, we focus on the most critical machine with unfixed edges and select the edge with the largest sum $OPA_{i,j}$ the direction with the largest $OPA_{i \to j}$ first.

To test this search heuristic we take the full propagation amount solver (fpa) and replace the search heuristic. The results of this observed full propagation amount search heuristic can be found in Table 5 under ofpa.

- ofpa performs better than fpa for 27 instances.
  - Notably, ofpa solves 4 instance that fpa didn't solve.
- ofpa performs equivalent to fpa for 2 instances.
- ofpa performs worse than fpa for 8 instances.

6.5.3 Kick-starting the observed full propagation amount using the full propagation amount

In this subsection we aim to answer the research question: Can we improve performance by using only observed full propagation amount values and some initialization similar to the impact based search heuristic by Refalo, 2004?

The observed full propagation amount search heuristic from the previous section has one potential downside. The first few choices are always made based on very little information. However, it is crucial to make good choices at the start of our search tree to reduce its total size.

Therefore, we implemented a new search heuristic that combines the full propagation amount and observed full propagation amount search heuristics. We combined these search heuristics as follows. We use the full propagation amount search heuristic until we reach a certain depth in the search tree. We experimented with a depth of 5. So, the first 5 choices in any search path are made using the full propagation amount search heuristic. From that point on we use the observed full propagation amount search heuristic. Note, that we do use the results from all applications of branching decisions to update our observed full propagation amount mapping.

As before we focus on the most critical machine with remaining edges, we select the edge with the largest sum $PA_{i,j}$ or $OPA_{i,j}$ and we explore the direction with the largest propagation amount first. As before, we took the full propagation amount (fpa) solver and replaced the search. The results of the new solver can be found in Table 5 under iofpa.

- iofpa performs better than fpa for 28 instances.
  - Notably, iofpa solves 2 instance that fpa didn't solve.
- iofpa performs equivalent to fpa for 1 instances.
- iofpa performs worse than fpa for 8 instances.

6.6 How important is the quality of the initial upper bound?

To answer this question we compare the critical solver and the ofpa solver when using the optimal makespan as the initial upper bound for each instance. The results of this experiment can be found in Table 6. The critical solver performs better in most cases. The difference in required time is at most a factor 10 (with one outlier). However, the ofpa solver can solve the instances "ft20" and "la40" where no other test from this research has been able to do that. This might suggest that the ofpa solver scales better to larger instances if it receives a strong initial upper bound.

6.7 How do our results compare with past results?

So far we only compared our own implementations. To check if we made any progress we will now compare the results of our best solver using the slack based search heuristic with the results where van der Sluis, 2022 uses the same slack based search heuristic. Note that, we are running our algorithms on a different computer with a CPU that has better single-thread performance according to cpubenchmark.net (Intel Core i7-10700 @ 2.90GHz vs AMD Ryzen 5 3600).

Therefore, we obtained the code from van der Sluis, 2022 and ran it on our computer. Table 7 contains the original results from van der Sluis, 2022 (their Table 7 under SC), the replicated results from running the same code on our computer and the best results from our algorithms when using the same search heuristic (Table 4 under critical).

One thing that can be observed is that there is no clear winner between the original results and the replicated results. We would have expected the replicated results to be better since it should be the same code running on a better processor. It is hard to really pinpoint why the results are so inconclusive since we can not be sure that the code that we received is in the exact same as the code that produced the original results.

However, for 30 out of the 37 our algorithm performs better. For 3 of those instances we find a solution where the algorithm by van der Sluis, 2022 could not. On the other hand there are 7 instances where we perform worse. In two of those instances he could find a solution and we could not.

instance	opt	basic solver	follow path	edge finding	partial graph	depth first	granular graph	flags	critical
abz5	1234	1253	1242	229.869	146.45	135.134	102.001	79.714	71.443
abz6	943	59.641	7.925	5.041	3.443	3.162	2.198	1.702	**1.838**
ft06	55	0.022	0.022	0.018	0.013	0.012	0.011	0.006	0.006
ft10	930	951	951	943	278.436	260.543	183.662	147.503	138.887
ft20	1165	1377	1267	1267	1267	1267	1267	1267	1267
la01	666	0.16	0.149	0.1	0.074	0.071	0.058	0.034	0.034
la02	655	3.949	2.922	1.81	1.264	1.192	0.924	0.601	**0.605**
la03	597	1.997	1.316	0.856	0.625	0.589	0.426	0.291	0.291
la04	590	5.145	0.994	0.644	0.472	0.446	0.347	0.238	0.234
la05	593	0.136	0.129	0.089	0.065	0.063	0.055	0.034	**0.036**
la06	926	0.755	0.717	0.443	0.32	0.304	0.251	0.179	0.179
la07	890	0.688	0.664	0.408	0.303	0.288	0.243	0.177	**0.179**
la08	863	0.734	0.709	0.406	0.313	0.29	0.225	0.154	**0.155**
la09	951	0.743	0.701	0.412	0.303	0.289	0.231	0.167	0.166
la10	958	0.788	0.774	0.452	0.331	0.319	0.269	0.176	0.176
la11	1222	2.387	2.295	1.256	0.929	0.891	0.746	0.537	**0.54**
la12	1039	2.412	2.355	1.255	0.918	0.877	0.742	0.51	**0.516**
la13	1150	2.232	2.189	1.19	0.868	0.836	0.696	0.51	0.509
la14	1292	2.016	1.927	1.057	0.778	0.751	0.637	0.477	**0.478**
la15	1207	1238	1208	1208	1208	1208	1208	1208	1208
la16	945	28.714	18.431	12.196	8.528	7.937	5.692	4.5	4.235
la17	784	0.345	**0.454**	0.305	0.246	0.216	0.137	0.099	0.099
la18	848	6.381	4.726	3.15	2.273	2.093	1.527	1.194	1.134
la19	842	16.903	16.426	10.59	7.014	6.632	4.606	3.764	**3.767**
la20	902	8.212	3.603	2.342	1.67	1.592	1.163	0.834	0.817
la30	1355	1388	**1404**	1404	1404	1404	1404	1404	1404
la40	1222	1295	**1314**	1295	1295	1295	1295	1295	1295
orb01	1059	1152	1152	1144	1116	1116	1103	1103	1098
orb02	888	74.198	34.288	22.748	15.908	14.691	10.26	8.195	**8.208**
orb03	1005	1052	1043	1043	1039	1039	1031	1031	1031
orb04	1005	244.109	97.138	65.064	46.826	43.775	29.655	23.595	23.55
orb05	887	915	150.427	103.043	73.441	69.147	49.856	40.571	37.782
orb06	1010	268.833	202.653	137.886	100.763	94.148	68.919	55.98	48.808
orb07	397	136.521	64.916	42.321	29.19	26.829	20.02	15.715	**15.72**
orb08	899	913	**918**	918	918	918	918	918	918
orb09	934	975	959	939	213.718	199.107	138.439	107.659	102.699
orb10	944	9.79	8.327	5.586	4.105	3.868	2.627	2.151	**2.241**

Table 4: Results of the first set of variants of our algorithm. We set a time limit of 5 minutes. We present the time in seconds required to find an optimal solution (as a real number) or the best solution found if the time limit is exceeded (as an integral number). A tilde indicates that a solver did not find any solution within the time limit. Note that each column (except opt) represents a solver that includes a new feature and all the features to its left. Results are bold if they are worse than the result directly to the left.

Algorithm 2 Edge finding

Require: $\mathcal{U}$: operations sorted by decreasing head
Require: $\mathcal{A}$: empty max tail heap
Require: $p_i^{+\mathcal{UA}} \leftarrow p_i$: remaining processing time
Require: $\mathcal{I}$: unique tail values sorted in increasing order
Require: $p_j^{+\mathcal{I}} \leftarrow \sum_{q_i=j} p_i^{+\mathcal{UA}}$: remaining processing time by unique tail value
Require: $t \leftarrow 0$: current time instant

while $0 < |\mathcal{U}| + |\mathcal{A}|$ **do** ▷ While $0 < |unavailableops + availableops|$
  while $0 < |\mathcal{U}| \wedge r_{\mathcal{U}.peek()} = t$ **do** ▷ Move each op c that release at t
    $c \leftarrow \mathcal{U}.pop()$
    $\mathcal{A}.push(c)$
    $P \leftarrow 0$
    for $j : reversed(\mathcal{I})$ **do**
      if $j = q_c$ **then**
        $break$
      end if
      if $0 < p_j^{+\mathcal{I}}$ **then**
        $P \leftarrow P + p_j^+$
        if $r_c + p_c + P + q_j > ub$ **then**
          $blocker_c \leftarrow j$
        end if
      end if
    end for
  end while
  if $|\mathcal{A}| = 0$ **then**
    $t \leftarrow r_{\mathcal{U}[-1]}$
    while $0 < |\mathcal{U}| + |\mathcal{A}|$ **do**
      while $0 < |\mathcal{U}| \wedge r_{\mathcal{U}[-1]} = t$ **do**
        $\mathcal{A}.push(\mathcal{U}[-1])$
        $\mathcal{U}.pop()$
      end while
    end while
  end if
  $i \leftarrow \mathcal{A}.peek()$
  $\epsilon \leftarrow p_i^+$
  if $0 < |\mathcal{U}|$ **then**
    $\epsilon \leftarrow \max(\epsilon, r_{\mathcal{U}[-1]} - t)$
  end if
  $t \leftarrow t + \epsilon$
  $p_i^+ \leftarrow p_i^+ - \epsilon$
  if $p_i^+ = 0$ **then**
    $\mathcal{A}.pop()$
  end if
end while

instance	opt	critical	fpa	ppa	ofpa	iofpa
abz5	1234	71.443	48.924	146.45	**26.73**	39.614
abz6	943	1.838	6.945	**3.443**	3.531	4.407
ft06	55	0.006	0.017	**0.013**	0.059	0.065
ft10	930	138.887	167.301	278.436	45.202	**23.87**
ft20	1165	1267	~	**1267**	~	~
la01	666	0.034	0.076	**0.074**	0.188	0.247
la02	655	0.605	18.949	**1.264**	2.186	3.296
la03	597	0.291	4.716	**0.625**	0.704	0.973
la04	590	0.234	5.353	**0.472**	0.799	1.104
la05	593	0.036	0.125	**0.065**	0.19	0.301
la06	926	0.179	1.404	**0.32**	0.723	1.016
la07	890	0.179	932	**0.303**	35.922	957
la08	863	0.155	2.077	**0.313**	0.695	1.45
la09	951	0.166	116.936	**0.303**	0.725	1.272
la10	958	0.176	1.465	**0.331**	0.681	1.164
la11	1222	0.54	14.628	**0.929**	2.517	7.681
la12	1039	0.516	33.87	**0.918**	2.699	4.469
la13	1150	0.509	4.901	**0.868**	130.048	4.467
la14	1292	0.478	6.824	**0.778**	1.9	3.908
la15	1207	1208	~	1208	~	**129.152**
la16	945	4.235	19.757	8.528	**2.482**	15.167
la17	784	0.099	1.544	**0.246**	1.033	0.99
la18	848	1.134	12.75	**2.273**	3.171	3.133
la19	842	3.767	17.338	7.014	8.785	**6.041**
la20	902	0.817	3.679	**1.67**	3.677	6.592
la30	1355	1404	1524	**1404**	~	~
la40	1222	1295	1306	**1295**	~	1314
orb01	1059	1098	1083	1116	**236.435**	1059
orb02	888	8.208	82.294	15.908	**17.537**	22.841
orb03	1005	1031	1099	**1039**	1040	1052
orb04	1005	23.55	129.653	46.826	38.881	**26.771**
orb05	887	37.782	280.393	**73.441**	78.644	84.469
orb06	1010	48.808	1013	100.763	**40.843**	108.804
orb07	397	15.72	21.736	29.19	32.823	**20.733**
orb08	899	918	940	918	**256.854**	1026
orb09	934	102.699	22.18	213.718	5.71	**4.234**
orb10	944	2.241	10.004	**4.105**	14.437	7.312

Table 5: Results of the second set of variants of our algorithm. We set a time limit of 5 minutes. We present the time in seconds required to find an optimal solution (as a real number) or the best solution found if the time limit is exceeded (as an integral number). A tilde indicates that a solver did not find any solution within the time limit. Note that each column (except opt) represents a similar solver which only varies in the search heuristic. The best result (excluding critical) for each row is in bold.

instance	opt	critical	ofpa
abz5	1234	31.848	**21.482**
abz6	943	**1.316**	2.285
ft06	55	**0.006**	0.055
ft10	930	31.598	**13.181**
ft20	1165	~	**10.401**
la01	666	**0.034**	0.173
la02	655	**0.064**	0.291
la03	597	**0.014**	0.179
la04	590	0.317	**0.207**
la05	593	**0.036**	0.173
la06	926	**0.188**	0.667
la07	890	**0.161**	1.079
la08	863	**0.16**	0.665
la09	951	**0.179**	0.687
la10	958	**0.181**	0.66
la11	1222	**0.55**	2.397
la12	1039	**0.523**	2.421
la13	1150	**0.525**	121.396
la14	1292	**0.491**	1.802
la15	1207	~	**6.237**
la16	945	2.263	**1.156**
la17	784	**0.035**	0.964
la18	848	**0.814**	1.024
la19	842	**3.695**	4.95
la20	902	**0.658**	2.798
la30	1355	~	~
la40	1222	~	**176.726**
orb01	1059	248.799	**184.561**
orb02	888	**6.794**	7.696
orb03	1005	**209.775**	1005
orb04	1005	**9.378**	15.345
orb05	887	**29.827**	50.237
orb06	1010	43.993	**14.952**
orb07	397	**6.783**	13.49
orb08	899	**4.621**	10.496
orb09	934	10.021	**1.623**
orb10	944	**2.254**	13.863

Table 6: Results of running the critical and ofpa solver with the optimal makespan as initial upper bound. We set a time limit of 5 minutes. We present the time in seconds required to find an optimal solution (as a real number) or the best solution found if the time limit is exceeded (as an integral number). A tilde indicates that a solver did not find any solution within the time limit. opt is the optimal makespan for each instance. The best result for each row is shown in bold.

instance	opt	original	replicated	critical
abz5	1234	1263	152.439	**73.458**
abz6	943	24.67	5.893	**1.749**
ft06	55	0.017	0.016	**0.007**
ft10	930	967	940	**155.801**
ft20	1165	1180	1226	1267
la01	666	0.126	0.103	**0.033**
la02	655	0.629	1.194	**0.437**
la03	597	3.92	4.268	**0.308**
la04	590	5.60	2.485	**0.312**
la05	593	0.232	0.195	**0.038**
la06	926	3.34	2.605	**0.175**
la07	890	11.89	10.629	**0.162**
la08	863	0.659	7.526	**0.143**
la09	951	5.837	3.960	**0.163**
la10	958	38.56	36.291	**0.154**
la11	1222	6.45	22.094	**0.461**
la12	1039	62.59	40.263	**0.463**
la13	1150	51.95	~	**0.478**
la14	1292	28.72	25.660	**0.448**
la15	1207	227.01	1232	1208
la16	945	979	8.955	**5.672**
la17	784	5.20	2.744	**0.142**
la18	848	75.49	26.086	**1.165**
la19	842	181.83	23.587	**3.833**
la20	902	916	912	**0.854**
la30	1355	~	1467	1404
la40	1222	1282	1294	1295
orb01	1059	1099	1065	1098
orb02	888	925	126.345	**8.571**
orb03	1005	1087	1073	**1031**
orb04	1005	192.518	59.324	**25.957**
orb05	887	906	889	**40.018**
orb06	1010	1071	282.727	**49.05**
orb07	397	409	251.222	**15.646**
orb08	899	934	91.007	918
orb09	934	939	40.254	122.467
orb10	944	3.54	2.508	**2.392**

Table 7: Best results compared to results from van der Sluis, 2022. In all cases the solver had a time limit of 5 minutes. The table presents the time in seconds required to find an optimal solution (as a real number) or the best solution found if the time limit is exceeded (as an integral number). A tilde indicates that a solver did not find any solution within the time limit. Results in the critical column are bold if they are better than the original and replicated results.

7 Conclusions

In conclusion, in this thesis we have aimed to develop an improved branching structure for constraint programming applied to the job shop problem.

We started with the description and implementation of a reference constraint programming system. Next, we presented several ideas that can be added to such a system to improve its performance. These ideas have been tested on a set of classical problem instances. Most of these instances had a size of up to 100 operations with varying numbers of jobs and machines. Only two instances had more than 100 operations (la30 and la40), however, not one of the presented methods could successfully solve either of those instances.

The first half of the the tested ideas focused on the constraint propagation part of the constraint programming system.

First, we showed that adding a follow path mode to a constraint programming system is beneficial. For a number of instances it halved the required time to find an optimal solution.

Next, we presented an alternative edge finding algorithm based on that by Carlier and Pinson, 1994. For some instances it allowed to shave off up to a third of the required time to find a solution when compared to the algorithm by Baptiste et al., 2001.

Then, we experimented with various graph propagation strategies. We showed that it helps to only propagate updates from recently affected parts of the disjunctive graph as opposed to updating through the entire disjunctive graph at once. Additionally, we showed that a depth first graph propagation approach works better than an algorithm based on finding a topological order first. After that we also tested an idea where we apply graph propagation more locally in the graph propagation loop after other propagation rules cause updates. These three ideas combined again allowed to half the required time to solve some instances (comparing the edge finding and granular graph solvers). This success seems to indicate that applying graph propagation more often and more locally works better than applying it only a few times globally.

Lastly, we explored a way to optimize the amount of times and the order in which we apply an edge finding algorithm during constraint propagation. This resulted in slightly better performance, but the improvement is not as dramatic as with the previous ideas.

The second half of the presented research focused more on alternative search heuristics that could improve on the full propagation amount search heuristic (fpa) by van der Sluis, 2022. We developed the following new search heuristics.

The partial propagation amount search heuristic (ppa) was aimed at saving time by using a less complex constraint propagation technique to determine the amount of propagation. In nearly all cases it performed better than the full propagation amount search heuristic.

The observed full propagation amount search heuristic (ofpa) inspired by Refalo, 2004 was aimed at saving time by approximating the expected full propagation amount based on results observed in the past. In most cases it performed better than the full and partial search heuristics.

The initialized observed full propagation amount search heuristic (iofpa) aimed to improve on this by using the full propagation amount search heuristic at the start of the search tree where it is crucial to make good branching decisions. However, the results from our experiments are inconclusive about its success.

Among these four search heuristics based on the propagation amount the observed full propagation amount (ofpa) search heuristic generally performed the best. It wasn't always the fastest, but it could solve two instances that none of the other three heuristics could. However, compared to the slack based search heuristic used in the critical solver there is still a lot of room for improvement.

To conclude our research we also tested how important the quality of the initial upper bound is. It is too early to draw general conclusion but it seems as though the observed full propagation amount search heuristics scales better to larger instances if the initial upper bound is strong enough.

7.1 Limitations

One limitation of the presented research is related to the first half of the presented ideas. For each new presented idea we take the solver including all the preceding presented ideas and evaluate the effect of adding the new presented idea. This means that we do not know the actual individual impact of each change and there might be some dependencies. For completion it would be nice to test each idea individually and/or all possible combinations of ideas.

One example of a possible dependency between multiple ideas is the following. Starting from the edge finding solver we first made a change to use graph propagation starting at recently affected operations to obtain the partial graph solver. Next, we replaced the graph propagation algorithm based on finding a topological ordering with a depth first approach to obtain the depth first. In both steps we observed improved performance. However, it might be the case that the depth first algorithm only performs better if we apply it to recently affected operations and not when we apply it to the entire graph. In other words, the second change might only help if we also include the first change.

The same can be said about the change made to obtain the granular graph solver. We applied graph propagation more locally after other propagation rules caused updates but we suspect that this only works well if the two ideas mentioned above in the previous paragraph are applied as well.

Additionally, this thesis only considered 2 larger instances and time limit of 5 minutes and many of the included instances were solvable within less than 1 minute. For a more general conclusion it might be important to test on larger instances as well. In that case the time limit should also be increased.

7.2 Future work

There are still many ways in which the presented constraint programming systems can be improved. Firstly, you can focus on implementing the $O(n \log n)$ edge finding algorithm by Carlier and Pinson, 1994 and presenting it in such

a way that it can become more widely adopted. Besides that, Katriel et al., 2005 present an algorithm for maintaining longest paths that could be used to improve the graph propagation part of the constraint programming system. Lastly, there might be better heuristics that can be used to predict the success of an application of an edge finding algorithm. Such heuristics could be used to reduce the number of useless edge finding calls.

References

Adams, J., Balas, E., & Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science*, *34*(3), 391–401.

Applegate, D., & Cook, W. (1991). Computational study of the job-shop scheduling problem. *ORSA journal on computing*, *3*(2), 149–156.

Balas, E., & Vazacopoulos, A. (1998). Guided local search with shifting bottleneck for job shop scheduling. *Management Science*, *44*(2), 262–275.

Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-based scheduling: Applying constraint programming to scheduling problems*. Kluwer Academic Publishers.

Błazewicz, J., Domschke, W., & Pesch, E. (1996). The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, *93*(1), 1–33.

Carlier, J. (1982). The one-machine sequencing problem. *European Journal of Operational Research*, *11*(1), 42–47.

Carlier, J., & Pinson, E. (1989). An algorithm for solving the job-shop problem. *Management Science*, *35*(2), 164–176.

Carlier, J., & Pinson, E. (1990). A practical use of jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research*, *26*, 268–287.

Carlier, J., & Pinson, E. (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, *78*(2), 146–161.

Cheng, C.-C., & Smith, S. F. (1997). Applying constraint satisfaction techniques to job shop scheduling. *Annals of Operations Research*, *70*, 327–357.

Clark, W. (1923). *The gantt chart: A working tool of management*. Ronald Press Company.

Crowston, W. B., Glover, F., Thompson, G. L., & Trawick, J. D. (1963). Probabilistic and parametric learning combinations of local job shop scheduling rules.

Da Col, G., & Teppan, E. C. (2019). Industrial size job shop scheduling tackled by present day cp solvers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *11802 LNCS*, 144–160.

Da Col, G., & Teppan, E. C. (2022). Industrial-size job shop scheduling with constraint programming. *Operations Research Perspectives*, *9*.

Dantzig, G. B., Orden, A., & Wolfe, P. (1945). The generalized simplex method for minimizing a linear form under linear inequality restraints. *A research memorandum*.

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Demirkol, E., Mehta, S., & Uzsoy, R. (1998). Benchmarks for shop scheduling problems. *European Journal of Operational Research*, *109*(1), 137–141.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of np-completeness*. Freeman.

Giffler, B., & Thompson, G. L. (1960). Algorithms for solving production-scheduling problems. *Operations Research*, *8*(4).

Glover, F., & Laguna, M. (1997). *Tabu search*. Springer New York.

Godard, D., Laborie, P., & Nuijten, W. (2005). Randomized large neighborhood search for cumulative scheduling. *ICAPS 2005 - Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 81–89.

Google. (2024). Or-tools [Accessed on May 9th]. https://developers.google.com/optimization

Graham, R., Lawler, E., Lenstra, J., & Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Discrete optimization ii* (pp. 287–326). Elsevier.

Granville, V., Rasson, J., & Krivánek, M. (1994). Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(6), 652–656.

Haupt, R. (1989). A survey of priority rule-based scheduling. *OR Spektrum*, *11*, 3–16.

IBM. (2024). Cp optimizer [Accessed on May 9th]. https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer

Jackson, J. R. (1955). Scheduling a production line to minimize maximum tardiness.

Jain, A., & Meeran, S. (1999). Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, *113*(2), 390–434.

Katriel, I., Michel, L., & Van Hentenryck, P. (2005). Maintaining longest paths incrementally. *Constraints*, *10*(2), 159–183.

Laborie, P., & Rogerie, J. (2016). Temporal linear relaxation in ibm ilog cp optimizer. *Journal of Scheduling*, *19*(4), 391–400.

Lawrence, S. R. (1984). *Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement)* (Doctoral dissertation). Graduate School of Industrial Administration (GSIA), Carnegie-Mellon University.

Martin, P., & Shmoys, D. (1996). *A new approach to computing optimal schedules for the job-shop scheduling problem* (Vol. 1084).

McMahon, G., & Florian, M. (1975). On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, *23*(3), 475–482.

Meiri, I. (1996). Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, *87*(1-2), 343–385.

Morrison, D. R., Jacobson, S. H., Sauppe, J. J., & Sewell, E. C. (2016). Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, *19*, 79–102.

Muth, J. F., & Thompson, G. L. (1963). *Industrial scheduling*. Prentice-Hall.

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

Nowicki, E., & Smutnicki, C. (2005). An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, *8*(2), 145–159.

Refalo, P. (2004). Impact-based search strategies for constraint programming. *Lecture Notes in Computer Science (including subseries Lecture Notes*

in Artificial Intelligence and Lecture Notes in Bioinformatics), *3258*, 557–571.

Roy, B., & Sussmann, B. (1964). Les problemes d'ordonnancement avec contraintes disjonctives. *SEMA, Note D.S.*, *9*.

Sadeh, N., & Fox, M. S. (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, *86*(1), 1–41.

Smith, S. F., & Cheng, C.-C. Slack-based heuristics for constraint satisfaction scheduling. In: 1993, 139–144.

Storer, R. H., Wu, S., & Vaccari, R. (1992). New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, *38*(10), 1495–1509.

Taillard, É. D. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, *64*(2), 278–285.

Taillard, É. D. (1994). Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, *6*(2), 108–117.

Vaessens, R. J. M., Aarts, E. H. L., & Lenstra, J. K. (1996). Job shop scheduling by local search. *INFORMS Journal on Computing*, *8*(3), 302–317.

van der Sluis, T. (2022). *Combining branch-and-bound and constraint programming for the job-shop problem* (Master's thesis). Department of Information and Computing Sciences, Utrecht University.

Van Laarhoven, P. J. M., Aarts, E. H. L., & Lenstra, J. K. (1992). Job shop scheduling by simulated annealing. *Operations research*, *40*(1), 113–125.

Vandevelde, A., Hoogeveen, H., Hurkens, C., & Lenstra, J. K. (2005). Lower bounds for the head-body-tail problem on parallel machines: A computational study of the multiprocessor flow shop. *INFORMS Journal on Computing*, *17*(3), 305–320.

van Hoorn, J. J. (2016). *Dynamic programming for routing and scheduling: Optimizing sequences of decisions* (Doctoral dissertation). VU University Amsterdam.

Vilím, P., Laborie, P., & Shaw, P. (2015). Failure-directed search for constraint-based scheduling. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9075*, 437–453.

Yamada, T., & Nakano, R. (1992). A genetic algorithm applicable to large-scale job-shop problems. *Parallel Problem Solving from Nature*, *2*, 281–290.

Zhang, C. Y., Li, P., Rao, Y., & Guan, Z. (2008). A very fast ts/sa algorithm for the job shop scheduling problem. *Computers and Operations Research*, *35*(1), 282–294.

A Edge finding analysis

In this appendix we will analyse the pseudo code of Algorithm 1.

The first and last three lines of the pseudo code are simply there such that we apply the changes after all possible changes have been determined and not whilst we are still running the algorithm.

The loop in line 4 fixes an operation k. It will be used to define sets of operations.

Next, the loop in line 6 fixes an operation i. These two operations define a set $\Omega_{i,k}$ which contains each operation j with $r_i \leq r_j$ and $d_j \leq d_k$. In other words, $r_{\Omega_{i,k}} = r_i$ and $d_{\Omega_{i,k}} = d_k$. Note that operation i is only part of this set if it has a deadline less than or equal to operation k (see line 7).

Lines 7-14 ensure that after each iteration of the inner loop we have that $P = p_{\Omega_{i,k}}$ and that $C_i = C = \max_{\emptyset \neq \Omega' \subseteq \Omega_{i,k}} r_{\Omega'} + p_{\Omega'}$.

Note that the inner loop iterates in decreasing order of release dates (since the operations are sorted in increasing order of release dates). This allows us to quickly update P and it allows the use of a rolling maximum to keep track of C (line 9).

Additionally, in lines 10-12 of the algorithm we can find the application of Equation 3e.

After the first inner loop we will have that $P = p_{\Omega_{1,k}}$ and that $C = \max_{\emptyset \neq \Omega' \subseteq \Omega_{1,k}} r_{\Omega'} + p_{\Omega'}$.

The loop in line 16 fixes an operation i again. If operation i is part of $\Omega_{i,k}$, then we update H and P (lines 17-20). This way we ensure that $P = p_{\Omega_{i,k}}$ stays true. Later, we will explain what H does.

If operation i is not part of $\Omega_{i,k}$ (because $d_i \leq d_k$ is false), then we attempt to apply Equation 3b. We can do this by substituting $\Omega = \Omega_{i,k}$ in the equation and O_i from the equation is simply operation i defined in line 16. Next we have that $r_{\Omega_{i,k} \cup O_i} = r_i$, $p_{\Omega_{i,k}} = P$ and $d_{\Omega_{i,k}} = d_k$. So the inequality in line 21 is the same as the one in the left-hand side of Equation 3b.

If the inequality holds, then we do not fix any edges explicitly but we do immediately apply Equation 3d (line 22). Note that C_i is exactly equal to the maximum taken in the right hand side of that equation. In essence, lines 21-23 check Equation 3b for the case where $r_{\Omega \cup O_i} = r_i$.

The variable H and lines 24-26 are used to check the case where $r_{\Omega \cup O_i} = r_j < r_i$. H is updated to maintain $H = \max_{j<i} r_{\Omega_{j,k}} + p_{\Omega_{j,k}}$. Therefore, $H + p_i > d_k = d_{\Omega_{j,k}}$ from line 24 is the same as the inequality in the left-hand side of Equation 3b. This can be seen by substituting $\Omega = \Omega_{j,k}$ in the equation whilst O_i still refers to operation i from line 16. Based on the definition of j, C and the values of each C_i we have that $C_j = C$, as seen in line 25.