Combining Branch-and-bound and Constraint Programming for the Job-Shop Problem

Master thesis written by:
Thimon van der Sluis (5489083)

(Under supervision of: Han Hoogeveen, Roel van den Broek and Marjan van den Akker.)

Department of Information Science
Utrecht University
August 22, 2022

Abstract

The Job Shop Problem is a well-studied scheduling problem which is proven to be NP-Hard. The Job Shop Problem consists of a set of jobs and a set of machines. For each job, one activity has to be processed on each machine and the order in which the activities have to be executed is predetermined for each job. The goal is to choose an ordering of activities for each machine such that the makespan is minimized. Approximation methods seem to work quite well at finding good solutions fast, but exact methods have trouble finding optima, especially for the bigger instances. In this thesis, we look at two exact methods for the Job Shop Problem, Branch-and-bound and Constraint Programming, and we test whether combining these two techniques reduces the amount of time needed to solve instances of the Job Shop Problem. To efficiently do this, we created new branching structures which make use of Constraint Programming in their choice of the next branch and then propagate the consequences of these choices. We found that adding Constraint Programming to a Branch-and-bound algorithm improves its efficiently by a lot, but calculating lower bounds in each node of a Constraint Programming algorithm has little effect and in some cases it even slows down the algorithm. A possible reason for this result is that both the edge-finding Constraint Propagation technique and the preemptive one-machine relaxation make use of the Jackson Preemptive Schedule in their calculations. Since the latter is used in cutting of branches of the search tree, it is possible that it is dominated by the Constraint Propagation techniques.

Contents

1 Introduction

The Job Shop Problem (JSP) is a well studied scheduling problem. The standard variant consists of scheduling a set of jobs on a set of machines, such that each of the jobs has an activity which has to be performed on each of the machines. The JSP is an optimization problem, the goal is to find a schedule such that all jobs have been scheduled and such that some objective is minimized (maximized). Usually this objective is to minimize the makespan of the schedule. The makespan of a schedule gets defined as the completion time of the last scheduled activity. Other objectives also get studied, including minimizing the maximal tardiness of a schedule or minimizing the total completion time of the activities.

Many techniques have been developed to find good (if not optimal) solutions for instances of the job shop problem. Good approximation techniques often involve Local Search methods like Simulated Annealing, Tabu Search or the Shifting Bottleneck method. Algorithms to find optimal solutions are mostly based on Branch-and-bound techniques [27]. Using efficient heuristics in selecting branches for Branch-and-bound helps guiding the algorithm to better solutions in less time. Another solution method concerns Constraint Programming. By modelling the JSP with an interval of start times for each activity, Constraint Propagation helps in narrowing these intervals and thus the search space.

In this thesis, the aim is to investigate the potential of combining Branch-and-bound and Constraint Programming. We want to find out how Constraint Programming and Branch-and-bound are used in the literature to solve the JSP and how they can be combined to compete with the state-of-the-art for solving the JSP. To do this, we compare Constraint Programming and Branch-and-bound hybrid algorithms with solely Branch-and-bound and solely Constraint Programming algorithms. We also create new branching schemes which use Constraint Programming in their choice for the next branch.

This thesis is organized as follows. In Section 2, a formal definition of the job shop problem is given, followed by a description of the *disjunctive graph* and then some notations which we will use in the remainder of this thesis. In Section 3, we give a description of different approximation methods for the job shop problem. In Section 4, we discuss different Branch-and-bound algorithms for the JSP. In Section 5, we explain Constraint Programming, followed by different Constraint Propagation techniques. In Section 6 the research questions for this thesis are introduced followed by the methodology in Section 7. In Section 8, we explain new branching structures that make use of Constraint Programming in their branching choices. In Section 9, a description of the implementation of the our solver is given. In Section 10 we describe the experimental settings and the experiments that we performed. In Section 11, the results of the experiments are given and in Section 12, we discuss these results.

This thesis has no explicit literature section, we discuss the literature within the section it corresponds to.

2 Job Shop Problem

An instance of the JSP consists of scheduling a set of n jobs $J = \{J_1, .., J_n\}$ onto a set of m machines $M = \{M_1, .., M_m\}$. Each of the jobs consists of one activity for each of the machines. Thus we have a total of mn activities $A_{1,1}, .., A_{1,n}, A_{2,1}, .., A_{m,n}$ (or A_i with $i \in \{1, .., mn\}$). For each activity A_i, the processing time p_i is known as well as to what machine and job it belongs. Within a job, there is a chain-like precedence constraints ordering $A_{i,1} \to A_{i,2} \to .. \to A_{i,m}$. The order of the activities on the same machine is yet to be determined. Furthermore, we denote O as the set of all activities, O_μ as the set of activities that have to be executed on machine $\mu \in M$ and O_j as the set of activities belonging to job $j \in J$.

The JSP can be represented as a graph. Each one of the precedence constraints for activities of the same job can be seen as a *directed edge (conjunctive constraint).* We can introduce an edge for each pair of activities that have to be performed on the same machine. This gives us *cliques* of *undirected edges (disjunctive constraints)* between activities on the same machine. A solution to the JSP is now equal to deciding a direction for each of the disjunctions such that there are no cycles in the graph. This results in the *disjunctive graph* $G = (O, U \cup D)$, where O is the set of activities, U is the set of edges corresponding to the conjunctive constraints and D is the set of edges corresponding to the disjunctive constraints. The weight of a conjunctive arc (i, j) is equal to p_i and the weight of a disjunctive arc (i, j) is equal to p_i or p_j, depending on its direction. The fictitious activities A_0 and A_* get added as a source and a sink of the disjunctive graph. They both have a processing time of 0. The makespan of a schedule can now be seen as the starting time of activity A_*.

A *schedule* on a disjunctive graph can be described by a set of starting times $S_i, \forall i \in \{1, .., mn\}$, such that all conjunctive constraints (2.1) and all disjunctive constraints (2.2) are satisfied [10]:

$$S_j - S_i \geq p_i, \qquad \forall (i,j) \in U, \tag{2.1}$$
$$S_j - S_i \geq p_i \text{ or } S_i - S_j \geq p_j, \qquad \forall (i,j) \in D. \tag{2.2}$$

Some papers in the literature consider the Preemptive Job Shop Problem (PJSP). In this variant of the JSP, an activity can be halted to perform another activity first. The PJSP is used as a relaxation of the JSP and it helps in determining lower bounds of the JSP.

2.1 Definitions

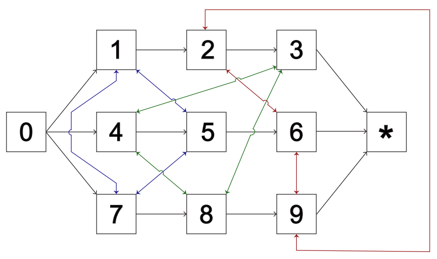
A *selection* S is a set of disjunctive arcs such that if $(A_i \rightarrow A_j) \in S$, $(A_j \rightarrow A_i) \notin S$. We can associate a conjunctive graph $G_S = (O, U \cup S)$ with this selection. A selection is *complete* if all disjunctions of D have been selected. It is *consistent* if there are no cycles in the graph. A schedule corresponds to a complete, consistent selection.

Let $S \subset D$ be a selection that denotes a set of directed arcs. Then $G_S = (O, U \cup S)$ is a directed graph that corresponds to the selection of S. If G_S is a complete and acyclic graph, then it defines a feasible schedule for the problem. The start time of activity A_i can be found by taking the longest distance from A_0 to A_i in this graph. The makespan of a solution can be obtained from a complete selection by taking the length of the longest path from A_0 to A_* [22].

In scheduling problems, there often are resources (machines) which are higher loaded than the others. We call these resources *critical resources* or *bottleneck machines*. According to Baptiste et al. [4], it is important to plan these in first because the makespan of the entire problem may be heavily influenced by these resources. (And otherwise the scheduling of critical resources may be delayed by the schedule of other less-critical resources.) We can find a critical machine by considering a one-machine relaxation to the problem; the machine with the highest lower bound is called critical.

When we find a schedule S (complete selection), its makespan is defined by the length of the longest path from A_0 to A_* using only the directed edges. We call any path of this length a *critical path*, if we want to find a better schedule something on this path must change. A sequence of two or more activities on the critical path that belong to the same machine is called a *critical block*. When a critical machine is found, we often look for a block or a path that makes this resource critical. Many of the local-search algorithms now consist of changing the order of the activities on the critical path and some branching structures also pick critical blocks and alter the order of their activities.

The one-machine subproblem is the problem we get when looking at only one machine. The other machines are relaxed to have infinite capacity. We only consider the activities on this one machine. An activity in the one-machine subproblem has a release date r_i, a processing time p_i and a tail q_i. When we model the problem as a disjunctive graph (with added source A_0 and sink A_*), the release date r_i of an activity A_i is defined as the length of the longest path from A_0 to A_i in the disjunctive graph (using only the directed edges). Its tail is defined as the longest path from A_i to A_* in the disjunctive graph, minus the processing time p_i [22]. Now, for any $K \subset J$, $H(K) = \min_{i \in K} r_i + \sum_{i \in K} p_i + \min_{i \in K} q_i$ is a lower bound to the optimal makespan of the one-machine problem. The optimal value to the one-machine subproblem is the maximum taken over all possible subsets $\max_{K \subset J} H(K)$. This value is a lower bound to solution of the JSP and it can be computed in $O(n \log n)$ time [9].



A_i	μ_i	r_i	p_i	q_i
A_1	1	0	1	4
A_2	3	1	2	2
A_3	2	3	2	0
A_4	2	0	3	6
A_5	1	3	3	3
A_6	3	6	3	0
A_7	1	0	2	4
A_8	2	2	1	3
A_9	3	3	3	0

Figure 2.1 & Table 1: *An example of a 3×3 disjunctive graph with added source (A_0) and sink (A_*). The coloured arcs denote the different machines and the black arcs denote the precedence constraints within the jobs.*

As an example, consider the disjunctive graph of Figure 2.1. We start with the precedence constraints (conjunctive arcs) $A_0 \to A_1 \to A_2 \to A_3 \to A_*$, $A_0 \to A_4 \to A_5 \to A_6 \to A_*$ and $A_0 \to A_7 \to A_8 \to A_9 \to A_*$. And we have the disjunctive arcs $\{(A_1, A_5), (A_1, A_7), (A_5, A_7)\}$ on Machine 1, $\{(A_3, A_4), (A_3, A_8), (A_4, A_8)\}$ on Machine 2 and $\{(A_2, A_6), (A_2, A_9), (A_6, A_9)\}$ on Machine 3, for which we have to choose a direction. The release dates of the activities on Machine 1 are $r_1 = 0$, $r_5 = r_4 + p_4 = 3$ and $r_7 = 0$. The tails of the activities are $q_1 = p_2 + p_3 = 3$, $q_5 = p_6 = 2$ and $q_7 = p_8 + p_9 = 4$. If we fix the disjunctions $A_1 \to A_5$ and $A_5 \to A_7$, then A_7 must come after A_1 (otherwise we would get a cycle in the graph). We then have $r_5 = \max(p_4, p_1) = 3$, $r_7 = \max(r_5 + p_5, r_1 + p_1) = 6$, $q_5 = \max(q_7 + p_7, q_6 + p_6) = 6$ and $q_1 = \max(q_2 + p_2, q_5 + p_5) = 6$.

Some of the papers from the literature use groups of activities Ω and constraints over these groups of variables. For this the following notation is used: p_Ω is the total processing time of the set of activities Ω, r_Ω is the earliest release date of all the activities in Ω, q_Ω is the smallest tail of all the activities in Ω, and d_Ω is the latest deadline in Ω.

2.2 Constraint Programming

For each activity A_i, the variable S_i is introduced, it denotes its start time. With r_i as the release date (earliest possible start time), and d_i as the deadline (latest possible end time) of A_i, $[r_i, d_i]$ is the window in which to execute activity A_i. The initial domain of S_i is $[r_i, lst_i]$ and activity A_i ends in the interval $[eet_i, d_i]$. Here, lst_i and eet_i are the latest start time and the earliest end time of activity A_i respectively. The release date r_i of an activity A_i equals the length of the longest path from A_0 to A_i in the disjunctive graph using only the directed edges. The deadline of an activity A_i is equal to a known upper bound ub minus the processing time p_i and the tail q_i of A_i: $d_i = ub - p_i - q_i$. We also have the relations $eet_i = r_i + p_i$ and $lst_i = d_i - p_i$. Note that after fixing a disjunction, the sets of successors and predecessors of certain activities may grow, resulting in smaller windows to execute the activities.

An instance of the Constraint Satisfaction Problem (CSP) is a triple $\mathbf{P} =< X, Y, C >$, where X is the set of all variables ($S_i \in X$), Y is the set containing the domains for every S_i, and C is the set of constraints. Each constraint is a rule between one or more start-variables. A solution to the CSP is an assignment of a value v_i to every start-variable S_i such that each v_i is in the domain of S_i and all constraints are satisfied.

Performing *Constraint Propagation* has two purposes: firstly, detecting if a partial solution at a given node can be extended into a complete solution with makespan lower than or equal to some upper bound ub. Secondly, reducing the domains of the start-variables S_i, which provides us with good information on which variables are the most constrained. Complex propagations take a lot of time which can be costly but they usually give better reductions for the intervals of the different start-variables than simple propagation techniques [4].

3 Approximation Methods

Many methods have been researched to solve the JSP. Local search methods like Simulated Annealing, Genetic Algorithms or Tabu Search have proven to be very good at finding good approximations of optimal solutions for the JSP. A lot of research has been done for exact methods like Branch-and-bound and Constraint Programming. Combinations of different methods have also been researched to create hybrid algorithms.

Local search methods like simulated annealing may rely on the quality of an *initial solution*. Furthermore, Branch-and-bound methods for the JSP use an initial solution to initialize an upper bound for the search tree. The initial solutions can be generated by various methods such as priority dispatching rules [23, 28], insertion algorithms [26, 32], the Shifting Bottleneck Procedure or by random methods.

3.1 Shifting bottleneck approach

The shifting bottleneck approach uses a basic form of iterated local search to produce substantially better schedules than were previously computed [18]. Adams, Balas and Zawack [1] present an example of the shifting bottleneck approach. First, they schedule the activities for each of the machines one at a time. Let $M_0 \subset M$ be the set of machines that have already been sequenced. For each machine not yet sequenced, they solve an one-machine relaxation (see Section 2.1) to optimality by determining the starting times of the activities on this machine. They use the outcome of the relaxation to find a *bottleneck machine*. The bottleneck is now the machine with the maximum optimal solution to the one-machine subproblem. The shifting bottleneck approach works as follows.

1. Identify the bottleneck machine ν among the machines $\mu \in M \backslash M_0$ and sequence the one-machine subproblem optimally. Set $M_0 \leftarrow M_0 \cup \{\nu\}$, go to (2).
2. Re-optimize the sequence of each critical machine $\mu \in M_0$ in turn, while keeping the other sequences fixed: set $M_0' = M_0 - \{\mu\}$ and solve the one machine subproblem for machine μ. Then if $M_0 = M$, stop, otherwise go to (1).

3.2 Tabu search

Tabu search uses a memory function (tabu list) to avoid being trapped at a local minimum. It was designed to find a near-optimum solution without (known) proof of convergence. Its performance is dependant of initial solutions, a good initial solution may lead to a good final solution [33].

In [19], Nowicki and Smutnicki explain their i-TSAB algorithm. i-TSAB is a tabu search algorithm which uses a set of elite solutions in its execution. The neighbours in this local-search algorithm consist of swapping activities in the critical path of a solution. The i-TSAB algorithm initializes a set of elite solutions by applying tabu search to random initial solutions. Then, the algorithm repeatedly modifies the set of elite solutions by performing tabu search, replacing the elite solutions with new ones. This is based on the fact that the global optima can be found at the center of a *big valley* and in each iteration we get closer to the center of this big valley [20].

Beck [6] introduced a tree-search algorithm for the JSP. The Solution-Guided Multi-Point Constructive Search (SGMPCS) was inspired by the i-TSAB algorithm but it uses a standard tree search with randomization combined with Constraint Propagation. Like the i-TSAB algorithm, it also uses a set of elite solutions.

Beck, Feng and Watson [7] proposed an algorithm that combines the i-TSAB algorithm with (a less complicated version of the) SGMPCS algorithm. It uses the fact that both of these algorithms use a set of elite solutions by passing along the set of elite solutions between the algorithms in every iteration.

3.3 Simulated annealing

Simulated annealing possesses a formal proof of convergence. Its behaviour is controlled by the cooling schedule. It may return to old solutions and oscillate between them since there is no memory used [33].

In [30], van Laarhoven et al. explain a simulated annealing approach for the JSP. Neighbours in this approach consist of schedules where successive activities on a critical path are reversed. This choice is motivated by two facts: (1) switching the order of a critical arc can never result in a cycle. (2) Switching the order of a non-critical arc can never result in a schedule with a lower makespan.

4 Branch-and-bound

Branch-and-bound is a solution approach that can be applied to a number of different types of problems. It is based on the principle that the total set of feasible solutions can be partitioned into smaller subsets of solutions. These subsets can be evaluated until a best solution is found. For the JSP, we can branch on fixing a disjunction in a certain direction or splitting the execution interval of an activity into two intervals. If we branch on a disjunction between activities A_i and A_j, the two subproblems we get are one where A_i comes before A_j and one where A_j comes after A_i. If we branch on splitting the execution interval of activity A_k, the two subproblems are one where the start time of the activity is lower than or equal to some value and one where it is higher than this value. Bounding is often done by considering a relaxation of the problem. A solution to this relaxation serves as a lower bound to the problem. If the best lower bound of a partial schedule in a node is not strictly better than the solution-value of an already known schedule, then the node does not have to be evaluated since its branch does not lead to an optimum. If this happens, we backtrack to the last node where getting an optimal schedule was still possible.

To create a good Branch-and-bound algorithm, a couple of questions have to be answered. What subproblems do we consider in each node? How do we backtrack when a node can not lead to an optimal solution? What bounds do we consider?

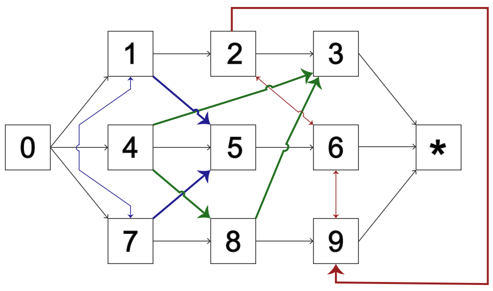
4.1 Branching schemes

Three different classes of branching schemes can be distinguished for the JSP. Either each node consists of choosing an ordering between a pair of activities, it consists of splitting the interval of the start time of an activity or we pick an activity A_i to be scheduled as early as possible and we fix all corresponding disjunctions in the direction $A_i \rightarrow A_j$.

4.1.1 Choosing an ordering

When we choose the ordering between a pair of activities, the heads and tails of some activities may change since the longest paths between activities in the disjunctive graph may change. The head of an activity A_i is always equal to the length of the longest path using directed edges only from A_0 to A_i and the tail of an activity is equal to the length of the longest path from A_i to A_* minus p_i. For instance, in the disjunctive graph of Figure 2.1, if we fix the ordering $A_4 \rightarrow A_8$, then we get $r_8 = \max(r_8, r_4 + p_4) = 3$ and $q_4 = \max(q_4, q_8 + p_8) = 3$. And because the longest path from A_0 now becomes $\{A_0, A_4, A_8, A_9\}$ with length 4, we get $r_9 = 4$.

In the branching structure of Colombani [14], we first choose the machine that has the strongest influence over the rest of the system, then we choose a disjunction on this machine for which we want to decide a direction. We use the maximum delay $\max_{\mu \in M} \delta_{neo_\mu}$



A_i	μ_i	r_i	p_i	q_i
A_1	1	0	1	6
A_2	3	1	2	3
A_3	2	4	2	0
A_4	2	0	3	6
A_5	1	3	3	3
A_6	3	6	3	0
A_7	1	0	2	6
A_8	2	3	1	3
A_9	3	4	3	0

Figure 4.1 & Table 2: *A disjunctive graph of a partial schedule and a table with the release date r_i, machine μ_i, processing time p_i and tail q_i for each activity A_i.*

over the set of not yet ordered activities *neo* to determine which machine is the most critical. Here, $neo_\mu \subset neo$ is the set of activities of machine μ for which not all disjunctions have been fixed and δ is defined as $\delta_J = r_J + p_J + q_J$ for a group of activities J. The choice of activity pairs to order is done by selecting on the selected machine a pair of activities from the neo set that minimizes the sum of the head of the first activity and the tail of the second: $\min_{(i,j)\in D}(r_i + q_j)$. In Figure 4.1, the *neo*-set consists of the activities A_1, A_2, A_6, A_7 and A_9. Thus we have $\delta_{\{A_1,A_7\}} = 0+1+2+6 = 9$ for Machine 1 and $\delta_{\{A_6,A_9,A_2\}} = 1+2+3+3+0 = 9$ for Machine 3. Both machines have the same maximum delay, so we pick one of them and we branch on the ordering with $A_i \to A_j$ for which $(r_i + q_j)$ is minimized. We pick Machine 3 and the ordering $A_2 \to A_6$ (because $A_2 \to A_9$ has already been chosen) and we keep $A_6 \to A_2$ as an alternative to try after backtracking.

The branching structure of Carlier and Pinson [11] is based on disjunctive arc pairs which define exactly two subtrees. Let i and j be any pair of activities which have to be scheduled on a critical machine with the highest one-machine lower bound. Then for both sequences of the two activities, we check if they increase the best lower bound LB. Let $d_{ij} = \max\{0, r_i + p_i + p_j + q_j - LB\}$, $a_{ij} = \min\{d_{ij}, d_{ji}\}$, $b_{ij} = |d_{ij} - d_{ji}|$.
The algorithm chooses a pair of disjunctive arcs which maximizes b_{ij} (maximize a_{ij} when tied) and creates a branch for $i \to j$ and one for $j \to i$.

We now assume that the Machine 3 in Figure 4.1 is critical and that the best known lower bound to the problem is $LB = 10$. Then for the pairs of activities (A_2, A_6) and (A_6, A_9), we calculate d_{ij}, d_{ji} and b_{ij}. For instance, $d_{2,6}$ is calculated as $d_{2,6} = \max(0, 1+2+3+0-10) = 0$. For the other values, we have $d_{6,2} = 4, d_{6,9} = 2$ and $d_{9,6} = 0$. Thus $b_{2,6} = 4$ and $b_{6,9} = 2$. Now, the pair of activities with the highest b_{ij} is (A_2, A_6). Thus we branch on this pair, creating one branch where $A_2 \to A_6$ and one branch where $A_6 \to A_2$.

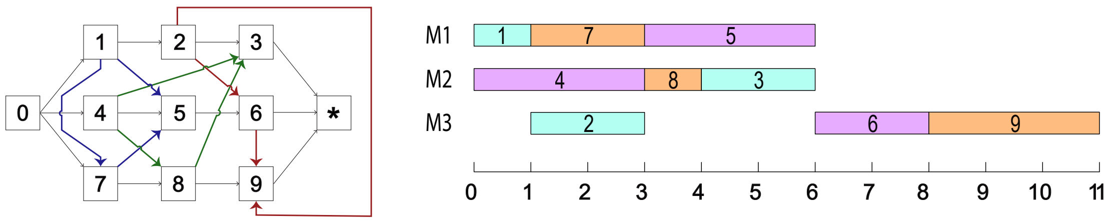


Figure 4.2: *Disjunctive graph and Gantt-chart of a complete schedule obtained from Figure 4.1.*

Barker and McMahon [5] associate with each node a complete schedule S heuristically. In practice, we can associate a complete schedule by using a heuristic. For instance, for the remaining disjunctions, we pick the activity A_i with the lowest $r_i + p_i$ and fix all the unfixed disjunctions from A_i to some activity A_j. In this complete schedule they find a critical activity (the earliest scheduled activity A_i where $r_i + p_i + q_i$ is at least the value of the best known solution) and a critical block (a sequence of two or more activities on the same machine which are on the critical path in S that ends in the critical activity). A set of subproblems is considered. In each of them, one of the activities in the critical block is made to precede all other activities of the block. The lower bound is an one-machine lower bound, computed for each machine. The branching continues in the node which has the lowest one-machine bound.

Brucker, Jurisch, and Sievers [8] also associate a complete schedule S with each node. Their branching algorithm consists of finding a critical path P in S and a critical block B on P. For each of the activities A_i in B, a branch is created for which the activity A_i is fixed to be processed before (after) all the other activities of B. A lower bound is calculated and if it does not exceed the known upper bound, we take the next activity of B and branch again (while A_i is still processed before (after) the other activities). If it does exceed the upper bound, A_i stays in its place in B and we take the next activity of B and branch again. See below for an example of the branching schemes of Barker and McMahon [5] and Brucker, Jurisch and Sievers [8].

From the example of Figure 4.1, we can obtain the complete schedule of Figure 4.2. The critical path $\{A_4, A_5, A_6, A_9\}$ contains only one critical block, which is $\{A_6, A_9\}$. The branching structure of [5] as well as the branching structure of [8] will now branch on the disjunctive arc between A_6 and A_9.

Smith and Cheng [25] branch on a disjunctive arc, chosen based on the "slack" of the disjunctive arcs. For two activities on the same machine A_i and A_j, the slack of an ordering is defined as

$$\text{Slack}(A_i \rightarrow A_j) = lst_j - eet_i. \tag{4.1}$$

Here $eet_j = r_j + p_j$ is the earliest time at which activity A_j can be finished if it starts after its release date, and $lst_i = d_i - p_i$ is the latest time on which activity A_i can be started if it ends before its deadline. If the slack of an ordering $A_i \to A_j$ is less than 0, then there is no schedule possible with this ordering (see Section 5.1.1). Thus, 4 cases can be identified:

1. (Slack($A_i \to A_j$) ≥ 0) and (Slack($A_j \to A_i$) < 0), then $A_i \to A_j$ is selected.
2. (Slack($A_j \to A_i$) ≥ 0) and (Slack($A_i \to A_j$) < 0), then $A_j \to A_i$ is selected.
3. (Slack($A_i \to A_j$) < 0) and (Slack($A_j \to A_i$) < 0), then an inconsistency is detected.
4. (Slack($A_i \to A_j$) ≥ 0) and (Slack($A_j \to A_i$) ≥ 0), then both orderings are still possible.

In cases 1 and 2, it is obvious what ordering we choose. In case 3, there are no feasible schedules, so we backtrack. For case 4, Smith and Cheng [25] research different choices in selecting what branch to explore first. In [13], Cheng and Smith use the function

$$\min_{(A_i,A_j)\in D}\{\frac{\text{Slack}(i\to j)}{\sqrt{S}}, \frac{\text{Slack}(j\to i)}{\sqrt{S}}\},$$

to determine what branch to explore first. Here

$$S = \frac{\min_{(A_i,A_j)\in D}\{\text{Slack}(i\to j), \text{Slack}(j\to i)\}}{\max_{(A_i,A_j)\in D}\{\text{Slack}(i\to j), \text{Slack}(j\to i)\}}.$$

Note that the first three cases of this branching structure are equivalent to performing disjunctive Constraint Propagation (Section 5.1.1).

Carlier and Pinson [11] created a branching scheme where the branching consists of picking an activity A_c on a certain machine and a set of activities K on the same machine. The function H (preemptive one-machine lower bound from Section 2.1) is defined as $H(I) = r_I + p_I + q_I$, for a set $I \subset J$. Here we can schedule every activity $A_i \in I$ freely. Similarly, we can have a bound of an ordering between an activity and a set of activities. If we fix A_c to be processed before all the elements K, then we have $H(A_c \to K) = r_c + p_c + p_K + q_K$, if A_c is to be processed after K, then $H(K \to A_c) = r_K + p_K + p_c + q_c$, and if A_c has to be processed within the set K, then $H(A_C \downarrow K) = r_K + p_K + p_c + q_K$. The branching structure finds a combination of an activity A_c and a set if activities K such that one of the following cases holds:

1. A_c can not be processed after K, but A_c can be processed within or before K. Then create a branch for $A_c \to K$ and create a branch for $A_c \downarrow K$.
2. A_c can be processed within K, but A_c can not be processed before or after K. Then we must have $A_c \downarrow K$.
3. A_c can be processed after or within K, but A_c can not be processed before K. Then create a branch for $K \to A_c$ and create a branch for $A_c \downarrow K$.

4. A_c can be processed before or after K, but it can not be processed within K. Then create a branch for $K \to A_c$ and create a branch for $A_c \to K$.

Note that this is very similar to performing the Not-first, Not-last propagation (Section 5.1.3).

In Table 3a, we have constructed an example of five activities that have to be processed on the same machine. The upper bound we use here is 15. We have $H(A_c \to K) = r_c + p_K + p_c + q_K = 4+7+3+2 = 16 > 15 = ub$, $H(A_c \downarrow K) = r_K + p_K + p_c + q_K = 2+7+3+2 = 14 < 15 = ub$ and $H(K \to A_c) = r_K + p_K + p_c + q_c = 2+7+3+0 = 12 < 15 = ub$. Thus A_c can not be processed before K and we create one branch where A_c must be processed after K and we create one branch where A_c must be processed before and after at least one activity of K (case 1). Examples of the cases 2, 3 and 4 can be found in Tables 3b, 3c and 3d respectively.

A_i	A_1	A_2	A_3	A_4	A_c
r_i	2	3	3	2	4
p_i	1	3	2	1	3
q_i	3	2	3	2	0

(a)

A_i	A_1	A_2	A_3	A_4	A_c
r_i	2	3	3	2	4
p_i	1	3	2	1	3
q_i	3	2	3	2	5

(b)

A_i	A_1	A_2	A_3	A_4	A_c
r_i	2	3	3	2	0
p_i	1	3	2	1	3
q_i	4	5	5	4	2

(c)

A_i	A_1	A_2	A_3	A_4	A_c
r_i	2	3	3	2	0
p_i	1	3	2	1	3
q_i	3	2	3	2	4

(d)

Table 3: *Examples for the four different cases of [11]. Here,* $K = \{A_1, A_2, A_3, A_4\}$ *and* $ub = 15$.

4.1.2 Assigning a start time

Baptiste, Pape and Nuijten [4] use a simple branching procedure with Constraint Propagation at each node of the search tree to determine whether the problem with makespan at most T has a solution. This is done in the following manner.

1) Select a machine on which the activities are not fully ordered.
2) Select an unscheduled activity that has to be processed on the chosen machine and make it start as early as possible. Use the propagation techniques to find more constraints. Keep the other activities as alternatives to be tried when backtracking.
3) Iterate step 2 until all activities on the chosen machine are ordered.
4) Iterate step 1 to 3 until all activities on all machines are ordered.

Nuijten and Aarts [21] give a constraint satisfaction approach based on search algorithms. Each node in the search tree corresponds to a partial solution and going from one node to another is done by assigning a start-time to an activity. The selection of a next activity and its start-time are done by selection-heuristics, one for the activity and then one for its start-time. Activities get selected by determining the activity A_i with the earliest minimal completion time $(r_i + p_i)$ of any unscheduled activity and then randomly selecting an activity that can be started before this minimal completion time (these are all the activities for which the processing-intervals overlap with that of A_i). After a start-time is assigned, inconsistent values of unassigned activities are removed. A start-time for an activity is inconsistent if there exists no feasible solution with this assignment. If by removing inconsistencies, the domain of a value becomes empty, a dead end is detected and we backtrack.

Florian, Trepant and McMahon [16] propose a Branch-and-Bound algorithm where at a certain time a subset of unscheduled activities is considered. Hereto, they use the "consecutive cut" dominance rule, where they pick the machine with the activity A_j from the cut C which can be finished the soonest (it has the lowest $r_j + p_j$ value). Here C is a cut of unscheduled activities for which all predecessors have been scheduled, thus each cut consists of one node per job. The machine μ with the activity A_j with the lowest $r_j + p_j$ gets selected and the start times of all activities in the cut on μ form the decision variables. The branching continues from the node with the smallest one-machine lower bound. When creating a node for an activity A_j, we schedule it to start at the earliest time possible, thus changing the release dates of the remaining activities on the same machine or job to be at least $r_j + p_j$. Note that this algorithm does not make use of the tails of activities, but we could create a symmetrical cut containing only the activities for which all the successors have been scheduled and similarly pick the activity with the lowest $q_j + p_j$ value.

Consider the following example. At the start of the tree search (see Figure 2.1), the cut consists of only the first activities of each job: A_1, A_4 and A_7. Of these activities, we pick the activity A_j with the lowest $r_j + p_j$; this is A_1. Then for all of the activities Ω within the cut that have to be performed on the same machine as A_1, we create a branch where we plan $A_i \in \Omega$ to start at its release date. Because A_7 is on the same machine as A_1, we create two branches, one where we have $A_1 \rightarrow A_5$ and $A_1 \rightarrow A_7$ and one where we have $A_7 \rightarrow A_1$ and $A_7 \rightarrow A_5$ (see Figure 4.3). Note that by using this branching structure we can omit parts of the search space ($A_5 \rightarrow A_7$ or $A_5 \rightarrow A_1$) without losing optimality. This is true because of the relation $r_5 \geq r_4 + p_4 \geq r_1 + p_1$, thus if we schedule A_5 to be processed at its earliest start time, then A_1 can be scheduled before A_5 without delaying it.

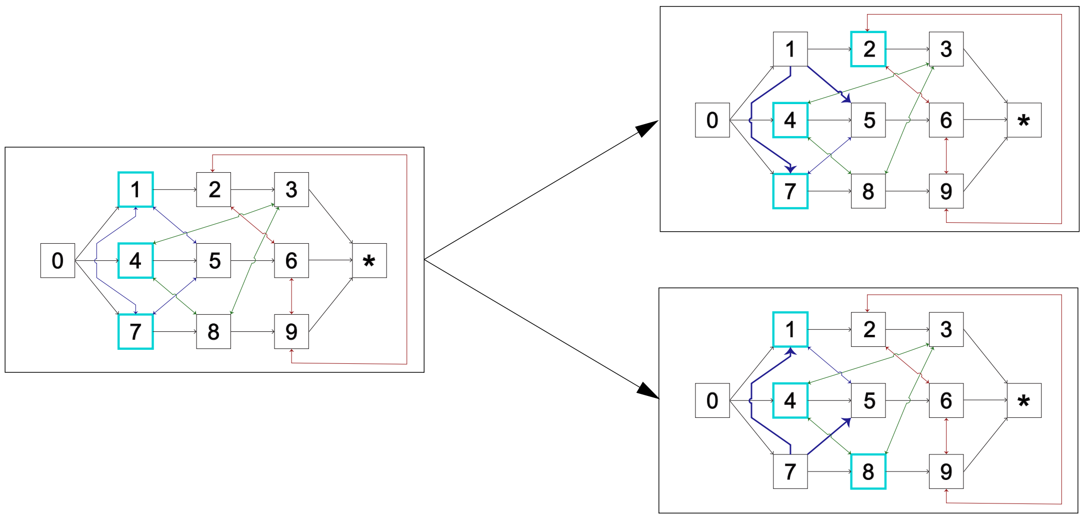


Figure 4.3: *Branching structure of Florian, Trepant and McMahon. The cuts are colored in cyan and the different machines have different colors (red, blue and green) for their arcs. Chosen arcs are bold.*

4.2 Bounds

The potential of branches in the Branch-and-bound method can get measured by making use of lower and upper bounds of these branches. The objective value of the best schedule found so far gets used as an upper bound. When we find a new, better schedule, the upper bound is altered. We can find lower bounds for each of the nodes in the search-tree by relaxing the problem. When we solve a relaxation and it gives us a lower bound for the node which is higher than or equal to the already found upper bound, we can stop at this node and backtrack; its children will never have a better objective value since they are subproblems of the current node.

If we pick any machine and allow preemption and then allow all other machines to have infinite capacity, then we get a relaxation for which the solution is a lower bound to the solution of the JSP. We can find the makespan of this relaxation by constructing Jackson's preemptive schedule (JPS). Jackson's preemptive schedule is the schedule associated with the most work remaining priority dispatching rule. The schedule is constructed by always executing the available, unfinished job with maximal tail. If we compute this bound for each machine, the we find a lower bound to the JSP with value $\max_{I \subset O_\mu} H(I)$ $(\mu \in M)$ [11]. Carlier and Pinson [9] state that there exists a subset $I \subset O_\mu$ such that either $H(I)$ equals the makespan of the non-preemptive optimal schedule f_0, or $f_0 - H(I) < p_c$ for some activity A_c of $O_\mu \backslash I$, then I is called a critical set and A_c is a critical activity. The edge-finding algorithm of Baptiste, Pape and Nuijten [4] constructs the JPS to update the heads of activities (Section 5.1.2). Because heads and tails are symmetrical, a Backwards Jackson's preemptive schedule (BJPS) can be constructed in a similar manner which can be used to update the tails of activities. The JPS and BJPS can both be computed in $O(n \log n)$ time.

Martin and Shmoys [18] give an ILP notation for the JSP which can be used to find lower bounds. For each job $j \in J$, $\mathcal{F}_j$ denotes the set of all job-schedules. A job-schedule contains an assignment of starting times to all of the activities of one job such that all of the precedence constraints are met and such that the job is finished by time ub. Let $x_{j\sigma}$ be a $0-1$ variable that indicates whether job j is scheduled according to job-schedule $\sigma \in \mathcal{F}_j$. Let $\gamma(\sigma, i, t)$ indicate whether job-schedule σ requires machine i at time t and let λ be the maximum number of jobs being processed on the same machine. M is the set of machines. Then the ILP formulation is to minimize λ subject to:

$$\sum_{\sigma \in \mathcal{F}_j} x_{j\sigma} = 1, \ \ \forall j \in J,$$

$$\sum_{j \in \mathcal{J}} \sum_{\sigma \in \mathcal{F}_j} \gamma(\sigma, \mu, t) x_{j\sigma} \leq \lambda, \ \ \forall \mu \in M, t = 1, .., T,$$

$$x_{j\sigma} \in \{0, 1\}, \ \ \forall j \in J, \sigma \in \mathcal{F}_j.$$

There is a feasible job-shop schedule which completes by time T if and only if the optimal value of the ILP equals 1. We can relax this to a LP by removing the binary constraints. Let λ^* be the optimal value of the relaxation, if we can show that $\lambda^* > 1$, then there is no schedule with makespan at most T and $T+1$ is a valid lower bound for the problem. The relaxation can be solved by using a combination of dynamic programming and the fractional packing algorithm of Plotkins, Shmoys and Tardos [24]. Martin and Shmoys [18] state that this algorithm finds better bounds than other bounds found by LP-relaxations, but that it is too slow to be useful in practice.

Brucker et al. [8] give lower bounds for when we put an activity on the edge of a block of activities. If an activity A_i in block B is moved in front of the block, all disjunctive arcs $\{(A_i, A_j)|A_j \in B$ and $j \neq i\}$ will be fixed in the direction $A_i \rightarrow A_j$. Thus,

$$r_i + p_i + \max\{\max_{A_j \in B, j\neq i}(p_j + q_j), \sum_{A_j \in B, j \neq i} p_j + \min_{A_j \in B, j\neq i} q_j\}$$

is a simple lower bound for the search tree node. And

$$\max\{\max_{A_j \in B, j\neq i}(r_j + p_j), \sum_{A_j \in B, j \neq i} p_j + \min_{j \in B, j\neq i} r_j\} + p_i + q_i$$

is a lower bound for the search tree node if activity i is moved to the very end position of block B. Promising subproblems are heuristically detected.

Vandevelde et al. [31] present different lower bounds for the Head-Body-Tail problem on parallel machines. These lower bounds were created for the multiprocessor flow-shop problem, but some of them can also be used for the Job Shop Problem. The *job-based bound* (JB) is the lower bound that we get by computing the minimal makespan for each job. Since this gives a lower bound for each of the jobs, taking the maximum over all of the jobs also gives a lower bound. We have:

$$\text{JB} = \max_{J_i \in J} r_{J_i} + p_{J_i} + q_{J_i}.$$

The set-based bound (SB) is the lower bound we get by considering the minimum idle time of each machine. Let $\bar{r}_i$ be the ith smallest heads. Then in the interval $[0, \bar{r}_1]$, all m machines are idle, in the interval $[\bar{r}_1, \bar{r}_2]$ at least $m-1$ machines are idle; until $\bar{r}_m$, which is the first possible moment when all machines can be operable. Thus we have at least $\sum_{j=1}^{m} \bar{r}_j$ idle time at the beginning of the schedule and similarly, we have at least $\sum_{j=1}^{m} \bar{q}_j$ idle time at the end of the schedule. Here $\bar{q}_i$ is the ith smallest tail. It follows that

$$SB = \frac{\sum_{j=1}^{m}(\bar{r}_j + \bar{q}_j) + \sum_{i=1}^{mn} p_i}{m}$$

is a lower bound for the current node of the search tree. This last bound can be calculated for any subset I of jobs with a size of at least m. The maximum over all these subsets is also a lower bound for the JSP and it is called the *subset-based bound* (SSB). Vandevelde et al. prove that $\max \text{SSB}, \text{JB}$ is bigger than the bound found by using the JPS, but the difference is less than or equal to $\frac{m-1}{m} p_{\max}$. Here $p_{\max}$ is the greatest processing time out of all activities.

5 Constraint programming

We can define a combinatorial optimization problem as an instance of a Constraint Satisfaction Problem (CSP) by formulating the problem as a decision problem and adding an upper bound for the problem solution. The CSP consists of a set of variables, a set of possibles values for these variables (domains) and a set of constraints between these variables. Constraints between variables say something about what combination of values are allowed for them. Constraints are either implicitly stated (e.g. arithmetic expressions) or explicitly stated (e.g. a set of tuples that satisfy the constraints). A feasible solution to the CSP is an assignment of values to the variables such that all the constraints are satisfied. Constraint Propagation is used to narrow down the search space of the variables [4]. With Constraint Programming, the aim is to find feasible solutions to the CSP.

As an example, we can consider a sudoku puzzle. The variables of a sudoku puzzle are the tiles in the 9×9 grid. The constraints of a sudoku puzzle are that on every row, column and subblock there should be exactly one of each of the numbers from 1 to 9. The domain of a tile are the numbers 1 to 9 that do not yet occur in a row, column or subblock of the tile. Now by logical reasoning, we can narrow down the values that each of the tiles can assume and by assigning a value to a tile and propagating its consequences, we can see if it is possible to find a solution with this assignment.

For the JSP, the variables are the starting times of the different activities. The constraints consist of the precedences between different activities on the same job (2.1) and of the orderings of activities on the same machine (2.2). The domains get modeled as a set of values within the interval $[r_i, lst_i]$. Because the JSP is an optimization problem, we can alter the CSP to an optimization problem by introducing some constraint on the outcome of the solution: $S_i + p_i \leq ub$ $(\forall A_i)$. If we can find an assignment of a value to each of the S_i's, then this will correspond to a solution of the JSP. Constraint Propagation consists of reducing the set of possible values for these variables by using information that is already known or by assuming information and then propagating the consequences [3]. The success of Constraint Propagation is due to the fact that the CSP needs all constraints to be met. If assigning a value to a variable leads to a contradiction, then the entire subset of solutions with this assignment can not contain a feasible solution.

Because CSP is a NP-complete problem, Constraint Propagation is usually incomplete. Thus it is beneficial to perform some kind of search to determine if a CSP has a solution or not. Traversing the search tree consists of two main things to think about: (i) How to go forward in the tree (what decision to make) and (ii) how to go backwards when finding a contradiction. The latter corresponds to adding additional constraints to the CSP and adjusting the domains of different variables. Constraint Propagation thus reasons on a combination of the original constraints and the new constraints which were added by making the decisions in the tree traversal [4].

A Constraint $c(v_1, .., v_n)$ is said to be arc-consistent if and only if for any variable v_i and any value x_i in the domain of v_i, there exist values $x_1, .., x_{i-1}, x_{i+1}, .., x_n$, such that $c(x_1, .., x_n)$ holds. Arc-consistency is the process of making sure that the domains of the variables match all of the constraints. For the JSP, the domain of a variable is often presented as a set of values within the interval, $[r_i, lst_i]$ (it can have gaps). Keeping arc-consistency on these bounds of the domain is called arc-B-consistency. Keeping arc-B-consistency is useful for problems like the non-preemptive JSP because updating the bounds is less costly than checking arc-consistency for each value inside the domains [4]. For the JSP, we can keep arc-B-consistency by updating r_i and lst_i for each activity A_i after fixing an ordering of two activities. We can find the gaps in the domain-intervals by keeping track of when each machine is occupied (and by what activity).

5.1 Propagation techniques

Constraint Propagation techniques might work well with Branch-and-bound since they allow us to keep the search tree small, which helps in speeding up the traversal of the Branch-and-bound tree, and they can help with finding better lower-bounds. By propagating certain constraints after fixing a disjunction, we may find some search-nodes that are not feasible. There are several methods to propagate constraints for a set of activities requiring the same machine. In this subsection, we discuss these techniques.

When we use a Constraint Propagation technique, we assume $r_i + p_i \leq d_i$ for every activity A_i. Otherwise there would not be a feasible solution possible and Constraint Propagation on some earlier node should have already found a dead end. The propagation techniques now try to find out if all the activities together can be processed in the time interval $[0, ub]$.

5.1.1 Disjunctive Constraint Propagation

If activities A_i and A_j have to be processed on the same machine, disjunctive Constraint Propagation consists of making sure activity A_i starts after A_j ends or A_j starts after A_i has ended:

$$[S_i + p_i \leq S_j] \vee [S_j + p_j \leq S_i].$$

Whenever the earliest end time of A_i exceeds the latest start time of A_j, A_i must be processed after A_j. Thus we add the constraint $S_j + p_j \leq S_i$ (or the ordering $A_j \rightarrow A_i$). When neither can proceed the other, a contradiction is detected.

In the example of Figure 5.1, two activities have to be performed on the same machine. A_1 can be processed within the interval $[0, 4]$ and A_2 can be executed within the interval $[1, 5]$. We have $lst_1 = 2$ and $eet_2 = 3$. Thus we can propagate the ordering $A_1 \rightarrow A_2$ and we put $lst_1 = 1$ and $r_2 = 2$.

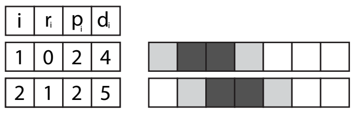


Figure 5.1: *Example of two activities on the same machine. The light grey tiles denote when the activities can be executed and the dark grey tiles denote the processing times.*

5.1.2 Edge-finding

Edge-finding consists of deducing that some activity A_i must, can, or cannot, execute before or after all the activities of a set of activities Ω. Edge-finding works as follows. Let r_Ω denote the smallest release date of the activities in Ω. Let d_Ω denote the largest deadline in Ω. Let p_Ω be the sum of the processing times of the activities in Ω. For any set of activities Ω not containing activity A_i, if the window of executing $[r_\Omega, d_{\Omega\cup\{A_i\}}]$ ($[r_{\Omega\cup\{A_i\}}, d_\Omega]$) is smaller than the combined processing time $p_i + p_\Omega$, then A_i must be executed before (after) Ω. And if an activity A_i comes after a set of activities Ω, then it must start (end) after (before) the earliest end time (latest start time) of any subset of Ω.

1. $\forall\Omega, \forall A_i \notin \Omega, [d_{\Omega\cup\{A_i\}} - r_\Omega < p_\Omega + p_i] \implies [A_i \ll \Omega]$,
2. $\forall\Omega, \forall A_i \notin \Omega, [d_\Omega - r_{\Omega\cup\{A_i\}} < p_\Omega + p_i] \implies [A_i \gg \Omega]$.

The algorithm that performs all the time-bound adjustments in $O(n^2)$ time is given in [4] and it uses the JPS in finding the edges. There is also a variant that uses more complex data-structures but runs in $O(n\log(n))$ time, given in [11]. And in [12], an edge-finding variant based on task intervals is given which runs in $O(n^3)$ time.

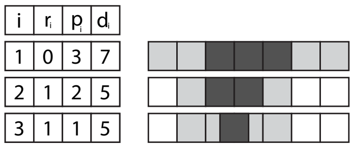


Figure 5.2: *Example of three activities on the same machine. The light grey tiles denote when the activities can be executed and the dark grey tile denote the processing times.*

In the example of Figure 5.2, three activities have to be performed on the same machine. The disjunctive Constraint Propagation does not find any changes in the domains of the activities. If A_1 is processed before or in between A_2 and A_3, then A_2 and A_3 can not both finish on time anymore. If we take $\Omega = \{A_2, A_3\}$, then the second edge-finding rule tells us that A_1 has to be performed after both A_2 and A_3, thus we can put $r_1 = 4$, $lst_2 = 2$ and $lst_3 = 3$.

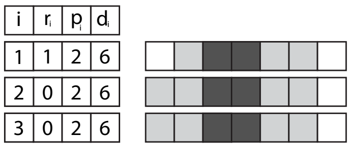
5.1.3 Not-first, Not-last

Opposite of the edge-finding rules, there are also rules to determine whether a certain activity can not be processed before (or after) a set of activities. If A_i comes before (after) the set of activities Ω, then the window of executing is given by $[r_i, d_\Omega]$ ($[r_\Omega, d_i]$), and if this is smaller than the combined processing time $p_{A_i \cup \Omega}$, then there exists no solution where A_i is executed before (after) Ω. This leads to the following rules:

1. $\forall \Omega, \forall A_i \notin \Omega, [d_\Omega - r_i < p_\Omega + p_i] \implies \neg(A_i \ll \Omega) \implies S_i \geq \min_{j \in \Omega} eet_j$,
2. $\forall \Omega \forall A_i \notin \Omega, [d_i - r_\Omega < p_\Omega + p_i] \implies \neg(A_i \gg \Omega) \implies S_i + p_i \leq \max_{j \in \Omega} lst_j$.

Here, the first rule is the not-first property and the second rule is the not-last property.

In [4], Baptiste, Pape and Nuijten give an algorithm which performs the time-bound adjustments corresponding to the Not-First rules. It runs in $O(n^2)$ time and $O(n)$ space. For the Not-Last rules, a symmetrical algorithm could be used which has the same time and space complexity.



i	r_i	p_i	d_i
1	1	2	6
2	0	2	6
3	0	2	6

Figure 5.3: *Example of three activities on the same machine. The light grey tiles denote when the activities can be executed and the dark grey tile denote the processing times.*

In the example of Figure 5.3, three activities have to be performed on the same machine. The disjunctive Constraint Propagation and the edge-finding rules do not find any changes in the domains of the activities. But, if we process A_1 before A_2 and A_3, then we can not process all three of the activities. If we take $\Omega = \{A_2, A_3\}$, then the Not-First rule propagates that A_1 can not be processed before Ω and we can change $r_1 = 2$.

5.1.4 Unnamed propagation

An activity A_i cannot start at a certain time t if there is a set $\Omega \subseteq I$ of activities with $A_i \notin \Omega$ and a u, $1 \leq u \leq |\Omega|$ such that at most $u - 1$ activities can be scheduled before t and at most $|\Omega| - u$ activities can be scheduled after $t + p_i$. We look for a *good combination* of A_i, Ω and u, such that:

$$(1 \leq u \leq |\Omega|, r_i < eet(\Omega, u) \wedge \ lst(\Omega, |\Omega| - u + 1) < \ eet_i) \implies r_i = \ eet(\Omega, u). \quad (5.1)$$

Here $eet(\Omega, u)$ ($lst(\Omega, u)$) is defined as the earliest end time (latest start time) of any subset $\Omega' \subset \Omega$ for which $|\Omega'| = u$. If the left hand side of equation (5.1) holds and we set S_i to be smaller than $eet(\Omega, u)$, then we still have $r_i < eet(\Omega, u)$ and we are again in the situation where we can only schedule $u - 1$ activities before A_i and $|\Omega| - u$ activities after A_i. Thus we can indeed put $r_i = eet(\Omega, u)$. This propagation can be done in $O(n^2 \log(n))$ time by using $eet(\Omega, u) = r_\Omega + \min_{\Omega' \subseteq \Omega : |\Omega'| = u} p_{\Omega'}$ (and similar for $\text{lst}(\Omega, u)$) [4].

5.1.5 Multi-machine propagation

Based on the idea that different machines are connected with each other through precedence constraints, Sourd and Nuijten [27] studied additional rules to propagate constraints between activities on different machines. The length of a path is defined as the sum of the processing times of the activities in the path, excluding the processing times of the two ends. If there is a path from activity A_i to activity A_j, then there is also a longest path between these two activities. We denote the length of this longest path by $l^0(A_i, A_j)$ (we set this to $-\infty$ if there is no path from A_i to A_j). This length serves as a lower bound on the time between the end of A_i and the start of A_j. Then for any set of activities Ω on a certain machine and for any activity A_j on a different machine, we have:

$$S_j \geq r_\Omega + p_\Omega + l^0(\Omega, A_j). \tag{5.2}$$

Here $l^0(\Omega, A_j) = \min_{A_i \in \Omega} l^0(A_i, A_j)$ is the minimal length of any path from an activity of Ω to A_j. Since all activities in Ω are to be scheduled on the same machine, the last activity of Ω cannot end before $r_\Omega + p_\Omega$. Because there now exists a path from Ω to A_j, we have that A_j cannot start before the end of the path from Ω to A_j.

5.1.6 Shaving

Shaving is a technique where an activity A_i is scheduled at the end or beginning of its domain and this fact is used to prove a contradiction or to reduce the domain of S_i. By trying out a starting time for an activity, and proving that this will not yield a feasible solution, we can omit it from the set of domains.

- Using the preemptive one-machine relaxation to determine that there is no feasible schedule is called *one-machine shaving.* It is equivalent to iterated Carlier-Pinson (C-P) (see [11]).
- Using the non-preemptive one-machine relaxation to determine that there is no feasible schedule is called *exact one-machine shaving.* It gives a slightly better bound than C-P but is significantly more costly.
- Using iterated Carlier-Pinson to show that there exists no schedule, is called *C-P shaving.*
- If we use C-P shaving to show that there is no schedule, this is called *double shaving*, it uses an extra recursion level of C-P. Double shaving gives very good bounds, but takes too much time to compute for it to be practical in a Branch-and-bound algorithm [18].

In [4], Baptiste, Pape and Nuijten explain a shaving algorithm, which works as follows. At each node of the search tree and for each activity A_i, the earliest time r_i at which the activity A_i can be scheduled without triggering a contradiction is computed. This consists of:

1. Iteratively trying a start time S_i for the activity A_i,
2. Propagating the consequence of this decision with the edge-finding techniques, and
3. Verifying that no contradiction has been detected. If a contradiction is found, remove the start time S_i from the domain of A_i.

By iteratively trying out start times for A_i, we shave off the parts of the domain of A_i for which there is no feasible solution.

6 Research questions

The goal of this thesis is to find out if the combination of Constraint Programming and Branch-and-bound is a good method for solving the (standard) Job Shop Problem. We also want to find out how different Branch-and-bound and Constraint Programming techniques compare to each other and in what way they should be combined for the best results. We do this by implementing and then comparing techniques from Section 4 and Section 5 of this thesis. This leads to the following research questions:

Q1. How to combine Branch-and-bound and Constraint Programming?

Q2. Does combining Branch-and-bound with Constraint Programming lead to a hybrid technique that works better than Branch-and-bound or Constraint Programming?

Q3. What kind of branch-choices/decisions should be made?

Q4. What are critical machines or activities and how do we find them?

Q5. What are relevant Constraint Propagation techniques and how do we use them?

7 Methodology

To answer the research questions of Section 6, we use the following method. First we implement a solver for the Job Shop Problem in python using a Branch-and-bound approach where each branch consists of either fixing the ordering of some heuristically chosen disjunction, splitting the start-interval of some activity in two or executing an activity at its (current) release date. To answer Questions 3 and 4, we will implement different Branch-and-bound algorithms for the JSP (from Section 4) and find out how they compare.

To answer Question 1, we want to come up with new branching structures where we branch on an activity or a pair of activities which is *close* to a propagation. For instance, if the domain of an activity A_i is one unit off of letting us use the Edge-finding propagation technique, then we would like to branch so that we can use this propagation. The goal is now to create a *measure of closeness* between an activity and a propagation, similar to how Cheng and Smith do this for the disjunctive Constraint Propagation in [25].

To answer Questions 2 and 5, we will implement the different Constraint Propagation techniques (Disjunctive Constraint Propagation and Edge-finding) and use Constraint Propagation on every node of the Branch-and-bound search tree in our implementation. We then compare how Constraint Programming influences the performance of a Branch-and-bound algorithm.

8 New branching schemes

To test the potential of combining Constraint Programming with Branch-and-bound for the JSP, we came up with new branching structures that make use of knowing that there will be Constraint Propagation after each branching choice. We expect that picking branches in which we have the largest amount of information, results in a relatively smaller search tree. We can measure the amount of information in a branch as a combination of disjunctions fixed and heads or tails increased. In this section, we explain the new branching schemes.

8.1 Slack-based branching

Smith and Cheng [13, 25] defined the slack of a ordering of two activities (Equation (4.1)). Similarly, we define the score of an ordering of two activities as

$$\text{Score}_{i,j} = r_i + p_i + p_j + q_j. \tag{8.1}$$

Here A_i and A_j have to be executed on the same machine. If the score of an ordering exceeds the upper bound for which we try to find a feasible schedule, the disjunctive Constraint Propagation technique will impose the opposite ordering. The branching structure of Smith and Cheng [25] (SC) now picks the ordering of any pair of activities which has the highest score and fixes it in the opposite direction. This is equivalent to fixing the ordering which is the closest to being imposed by the disjunctive Constraint Propagation technique.

8.1.1 Increasing head or tail based on slack

If we select the ordering of a pair of activities with the highest slack, another way of inducing the ordering $A_j \to A_i$ is by increasing the head r_i of activity A_i (or the tail q_j of activity A_j) and then letting the disjunctive Constraint Propagation impose the ordering of the pair. The idea is that when we are close to disjunctive Constraint Propagation, then in the first branch we get the ordering because of the propagation and in the second branch we get a start-interval which is as big as the difference between ub and the score of the ordering.

Suppose that the Constraint Propagation techniques find every ordering $A_p \to A_q$ for which the score $\text{Score}_{p,q} >$ ub and fixes the disjunction in the right direction. Let $A_i \to A_j$ be the ordering with the highest score which was not fixed by the Constraint Propagation, then we have that $\text{Score}_{i,j} = r_i + p_i + p_j + q_j \leq ub$. If we can get the score of this ordering to exceed the upper bound, then the propagation will impose the opposite ordering. To do this, we increase the head of A_i (or tail of A_j): $r'_i = r_i + ub - \text{Score}_{i,j} + 1$. After increasing the head of activity A_i, its start-interval $[r_i, \text{lst}_i]$ splits into two intervals, $[r'_i, \text{lst}_i]$ and $[r_i, r'_i - 1]$. When we backtrack and go into the opposite branch, we thus change the tail $q'_i = ub - p_i - r'_i + 1 = p_j + q_j$.

Theorem 1. *If $r_j + 2p_j + q_j < ub - p_i + 1$, then increasing the head of activity A_i gives more information than (SC) would give in the first branch. Increasing the tail of the activity in the second branch gives less information than (SC) gives.*

Proof. After increasing the head of A_i to $r'_i = r_i + ub - \text{Score}_{i,j} + 1$ we get $A_j \to A_i$ because of the disjunctive Constraint Propagation. In the branching structure of Smith and Cheng, we get that $r''_i = \max(r_i, r_j + p_j)$ and $q''_j = \max(q_j, q_i + p_i)$. Thus if $r_i + ub - \text{Score}_{i,j} + 1 > r_j + p_j$, the new branching structure gives more information. This is equivalent to: $ub - p_i + 1 > r_j + 2p_j + q_j$.

In the opposite branch, Smith and Cheng fix the ordering $A_i \to A_j$, causing $r''_j = \max(r_j, r_i + p_i)$ and $q''_i = \max(q_i, q_j + p_j)$. In the new branching structure, we get $q'_i = q_j + p_j$. For the new branching structure to gain as much information in the second branch as S-C, it has to propagate the ordering $A_i \to A_j$. This happens when $r_j + p_j + p_i + q'_i > ub$, which is equivalent to $r_j + p_j + p_j + q_j \geq ub - p_i + 1$. □

We refer to the branching structure where we increase the head or tail of an activity based on the slack as *Decrease Slack Branching (DSB).*

Theorem 2. *If $q_i \geq q_j + p_j$, then:*

- *The branch where we increase r_i leads to a second branch that is equal to its parent.*
- *The branch where we increase q_j leads to a second branch where $r'_j = r_i + p_i$.*

Proof. When going into the second branch after increasing the head r_i of activity A_i in the first branch, we increase q_i to $\max(q_i, q_j + p_j)$. If we have that $q_i \geq q_j + p_j$, then the new value of q_i equals its old value. Thus, in the second branch of this branching scheme, nothing changes and we are in the same situation as its parent. But, since we choose the pair of activities for which $\text{score}_{i,j} = r_i + p_i + p_j + q_j$ is maximal, if we increase q_j instead of r_i in the first branch, such that the score goes over the upper bound, this will also impose the ordering $A_j \to A_i$. If we were to have $r_j \geq r_i + p_i$, then: $\text{score}_{i,j} = r_i + p_i + p_j + q_j \leq r_j + q_i < r_j + p_j + p_i + q_i = \text{score}_{j,i}$. This contradicts the choice of $\text{score}_{i,j}$ being maximal. Thus we must have $r_j < r_i + p_i$ and when going into the second branch, we have $r_j = r_i + p_i$. □

Theorem 3. *Let $(A_i, A_j) \in D$ be the pair of activities with the highest score $Score_{i,j}$. If $q_i \geq q_j + p_j$, then:*

- *$r_i \geq r_k + p_k$ for all A_k with $(A_k, A_i) \in D$,*
- *and $q_i \geq q_k + p_k$ for all A_l with $(A_k, A_i) \in D$.*

Proof. For all activities A_k for which there is no directed arc between A_k and A_i, we have:

$$r_i + p_i - r_k - p_k \geq q_i + p_i - q_j - p_j \geq p_i.$$

It follows that we have $r_i \geq r_k + p_k$ for all activities A_k for which $(A_k, A_i) \in D$.
For all activities A_l for which there is no directed arc between A_k and A_i, we have:

$$r_i + p_i + q_i \geq r_i + p_i + p_j + q_j \geq r_i + p_i + p_k + q_k.$$

It follows that we have $q_i \geq p_k + q_k$ for all activities A_k for which $(A_k, A_i) \in D$. $\square$

When using the DSB branching scheme, we check if $q_i < q_j + p_j$ and if it does, we branch on splitting the start interval of A_j in two. Because of Theorem 2, branching on increasing the head or tail of an activity (DSB) is a valid branching strategy; it always results in a left branch and a right branch where some constraint is added to the solution in both branches. Because of Theorem 3, if we find a pair of activities (A_i, A_j) with maximal score $\text{Score}_{i,j}$ for which $q_i \geq q_j + p_j$, we have that A_i has the biggest head and the biggest tail out of all activities on the same machine for which there is no directed arc to or from A_i. Therefor, A_i might be the reason why its machine is critical and we thus think it is a good idea to fix direction from or to A_i first.

A_i	A_1	A_2	A_3
r_i	5	2	3
p_i	1	2	1
q_i	5	3	2

Table 4: *Example of three activities for the same machine.*

Consider the activities of Table 4 and let the upper bound be 11. $\text{Score}_{1,2} = 11$ is the highest score of all the pairs of activities. If we increase r'_1 to 6 or q'_2 to 4, then the disjunctive Constraint Propagation will impose the ordering $A_2 \to A_1$. Because we have $q_1 = 5 = q_2 + p_2$, we choose to branch on increasing q_2. It follows immediately that A_1 should be processed within the interval $[5, 6]$.

8.1.2 Fixing disjunctions

Instead of directly increasing the head or tail of an activity A_i, we can also look for an ordering $(A_k \to A_i)$ which increases the head of A_i enough for $\text{Score}_{i,j}$ to get bigger than the upper bound. For this to happen, we need to find an activity A_k on the same machine as A_i and A_j for which we have $r_k + p_k > r_i + ub - \text{Score}_{i,j}$ (or $q_k + p_k > q_j + ub - \text{Score}_{i,j}$). In the opposite branch, we would fix the ordering $A_i \to A_k$.

It is possible that there is no disjunction between A_i and another activity for which we can fix an ordering to get $\text{Score}_{i,j}$ over the upper bound. In this case we can choose a disjunction on a different machine for which one of the activities is a predecessor of A_i. Let (A_k, A_l) be such a disjunction for which A_l is a predecessor of A_i and belongs to the same job. We fix $A_k \to A_l$ if it increases the head r_i such that $\text{Score}_{i,j}$ goes over the upper bound. Symmetrically, if there is an ordering of a disjunction between a successor A_k of A_j and another activity that will increase the tail of A_j, fixing this ordering will result in propagating $A_j \to A_i$ as well. We can keep taking predecessors within the job of A_i until we find a disjunction between a predecessor and A_i which will increase r_i (or q_j) enough upon fixing the disjunction.

If there are no orderings available which will cause $\text{Score}_{i,j}$ to go over the upper bound, we simply branch on fixing $A_j \to A_i$.

We call the branching structure where we fix a disjunction on the same machine which increases the score of an ordering *Machine Disjunction Slack Branching (MDSB).* The branching structure where we look for a disjunction with an activity of the same job is called *Job Disjunction Slack Branching (JDSB).*

8.2 Edge-finding score

Similarly to how the slack of Smith and Cheng is a measure of how close an ordering is to being propagated by the Disjunctive Constraint Propagation, we can define closeness to performing Edge-Finding by introducing the score of a combination of an activity and a set of activities:

$$\text{Score}_{c,\Omega} = r_{\{A_c\}\cup\Omega} + p_c + p_\Omega + q_\Omega, \tag{8.2}$$

$$\text{Score}_{\Omega,c} = r_\Omega + p_c + p_\Omega + q_{\{A_c\}\cup\Omega}. \tag{8.3}$$

If this score goes over the upper bound for which we want to find a schedule, we know that the Edge-finding propagation technique will determine that A_c should be processed after (before) all of the activities of Ω. Thus, in each node, we want to create a branch where we either increase $r_{\{A_c\}\cup\Omega}$ or $q_{\{A_c\}\cup\Omega}$ for the right combination of A_c and Ω.

There are two cases depending on which activity has the smallest head: A_c or some $A_i \in \Omega$. If $r_c < r_i = r_\Omega$, then we can increase r_c by $r_\Omega - r_c$ for it to still be the smallest element of the set $\{A_c\} \cup \Omega$. We can increase r_c further such that $\text{Score}_{c,\Omega} > ub$ if and only if $r_\Omega + p_c + p_\Omega + q_\Omega > ub$. The activity A_c is executed after every activity of the set Ω if its head r_c goes over $ub - p_\Omega - p_c - q_\Omega$. Alternatively, if there is an activity $A_i \in \Omega$ with a smaller head than r_c, then the amount we can increase r_i for it to still be the smallest head is $r_j - r_i$. Here $r_j = r_{\{A_c\}\cup\Omega\setminus\{A_i\}}$ is the second smallest head of the set $\Omega \cup A_c\}$. Thus we have that if $r_{\{A_c\}\cup\Omega\setminus\{A_i\}} + p_c + p_\Omega + q_\Omega > ub$, then r_c increases to $r_\Omega + p_\Omega$.

Theorem 4. *Let $\Omega = \{A_1, A_2, .., A_k\}$ be a set of activities that is ordered such that $r_1 \leq r_2 \leq .. \leq r_k$. For an activity $A_c \in \Omega$, if $r_2 - r_1 \geq ub - Score_{c,\Omega} + 1$, then we can increase r_1 enough for the Edge-Finding Propagation to determine that A_c has to be processed after all of the activities in Ω.*

Proof. If $r_2 \geq r_1 + ub - \text{Score}_{c,\Omega} + 1$, then we can increase r_1 by $ub - \text{Score}_{c,\Omega} + 1$ without losing that r_1 is the smallest head of Ω. If we do increase r_1 with this amount, then we get that $\text{Score}_{c,\Omega} = ub + 1 > ub$ and thus the Edge-Finding Propagation will put A_c after Ω. □

Theorem 5. *For a given activity A_c and a given set of activities Ω, let A_j be the activity with the smallest head of $\Omega \cup \{A_c\}$. Furthermore, let $x = ub - score_{c,\Omega} + 1$ and let A_i be the activity with the smallest head of $\Omega \cup \{A_c\} \backslash \{A_j\}$ greater than or equal to $r_j + x$. If there is an activity in $\Omega \cup \{A_c\} \backslash \{A_j\}$ with a smaller head than A_i, then if $q_c \geq q_\Omega$ or if $r_i \geq r_j + x + p_c$, then one of the activities $A_k \in \Omega \cup \{A_c\}$ with $r_k < r_i$ should be executed before all of the other activities of $\Omega \cup \{A_c\}$.*

Proof. If all activities A_k with $r_k < r_j + x$ are executed at the end of Ω, then $r'_\Omega = r_i \geq r_j + x$. The last activity of $\Omega \cup A_c$ is either A_c or some $A_b \in \Omega$. If the last activity is A_b, then we have $r'_\Omega + p_\Omega + p_c + q_b \geq r_j + x + p_\Omega + p_c + q_\Omega > ub$. If the last activity is A_c and $q_c \geq q_\Omega$, then we have $r'_\Omega + p_\Omega + p_c + q_c \geq r_j + x + p_\Omega + p_c + q_\Omega > ub$. If the last activity is A_c and $r_i \geq r_j + x + p_c$, then we have $r'_\Omega + p_\Omega + q_\Omega \geq r_j + x + p_c + p_\Omega + q_\Omega > ub$. Thus at least one of the activities with a head smaller than $r_j + x$ should be executed first. $\square$

Note that in practice, we never have $q_c \geq q_i = q_\Omega$, because if we do we can choose $\Omega' = \Omega \cup \{A_c\} \backslash \{A_i\}$ and we have $\text{Score}_{i,\Omega'} \geq \text{Score}_{c,\Omega}$. Because of Theorem 5, if we find an activity A_c and a set of activities Ω with high $\text{Score}_{c,\Omega}$ which is being "blocked" by one or more activities, then if r_i if bigger than or equal to $r_j + x + p_c$, then we can create a branch where we put one of the activities of $\{A_b \in \Omega \cup \{A_c\} : r_b < r_j + x\}$ in front of the other activities of $\Omega \cup \{A_c\}$. This is especially useful if there is only one activity A_k for which $r_j < r_k < r_j + x$ because then we can create one branch where we execute A_k before all of the activities of Ω and one branch where we execute A_j before all of the activities of Ω since they can not both be executed after Ω. When there are more than one *blocking activities*, then we can create a similar branching structure but the node will have more than two branches.

Table 5: *An example for calculating the edge-finding score (a) and an example for when the head of an activity is being blocked by another activity (b).*

A_i	A_1	A_2	A_3	A_4	A_5
r_i	4	1	3	3	3
p_i	1	8	1	3	1
q_i	3	4	4	4	4

(a)

A_i	A_1	A_2	A_3	A_4	A_5
r_i	4	1	3	1	3
p_i	1	8	1	3	1
q_i	3	4	4	4	4

(b)

As an example, we look at the activities in Table 5a. The upper bound for these examples is 19. The activities all belong to the same machine. The score $\text{Score}_{1,\{A_2,A_3,A_4,A_5\}} = 1 + 14 + 4 = 19$ is the highest score of all combinations of activities and sets of activities. The activity with the smallest head is A_2 with $r_2 = 1$. It needs to increase to 3 for the edge-finding to be able to put A_1 after $\{A_2, A_3, A_4, A_5\}$. Because the activity with the second smallest head has head 3, we can increase r_2 to 3 and have it still be the smallest head. We now look for a disjunction which pushes r_2 over 3 and if we can not find such a disjunction, we branch on splitting the domain of the start interval S_2: in the first branch we have $r_2 = 3$ and in the second branch we have $q_2 = 9$ to enforce that A_2 starts at time 2 at the latest.

In Table 5b, r_2 is being “blocked” by r_4, so if we increase r_2 to 3, then r_4 will have the smallest head. The second smallest head is $r_3 = r_5 = 3$, thus if A_3 (or A_5) were to start before all the other activities, then we would get $r_3 + p_\Omega + q_1 = 20 > 19 = ub$. Thus, we create a branch where A_2 is the first activity to be processed and if this branch leads to a dead end, we create a branch where A_4 is the first to be processed.

Instead of looking at the combination of an activity and a set of activities that is closest to Edge-Finding, we can also find the combination of an activity and a set of activities for which the Edge-Finding will increase the head or tail the most upon increasing $\text{Score}_{c,\Omega}$ to go over ub, or we can look for a combination of being close to propagation, the amount of propagation we get and the changes that follow.

8.2.1 Increasing head or tail to impose Edge-Finding

To make the score of a set of activities Ω and an activity $A_i \notin \Omega$ go over the upper bound, we can directly increase the head or tail of an activity and propagate the consequences. We call this branching scheme *Increase Edge-Find Branching (IEB).*

8.2.2 Fixing disjunctions to impose Edge-Finding

Instead of directly increasing the head or tail of an activity A_i such that the score of an ordering goes over the upper bound, we can also look for a disjunction which, when given a direction, increases the head or tail of A_i enough to go over the current upper bound. To find such a disjunction, we can:

- Search within the disjunctions of the machine where A_i has to be scheduled on. If we find a disjunction which will increase r_i or q_i enough then we can choose to fix this disjunction in the right direction. We call this branching scheme *Machine Disjunction Edge Branching (MDEB).*
- If there is no disjunction within the same disjunction of the machine of A_i, we can take a predecessor (successor) of A_i within the same job and try to find an ordering of a disjunction which will increase the head (tail) of this predecessor (successor) enough. We call this branching scheme *Job Disjunction Edge Branching (JDEB).*

It is possible that there are no orderings of disjunctions between predecessors of the activity for which we want to increase the head. In this case, we branch by using the DSB branching scheme. It is also possible that there is no combination of activity and set of activities for which we can increase the head (or tail) such that the score goes over the upper bound because the second smallest head (or tail) is too small, in this case we do not find a branch with this branching scheme. In practice, we will then use the slack-based branching scheme (SC) to find a new ordering between a pair of activities on a critical machine (highest lower bound).

8.3 Selecting an activity

Florian, Trepant and Mcmahon [16], branch by selecting an activity A_i for which the predecessors within the same job have all been processed and they schedule it as early as possible. Then every disjunction between A_i and some other activity A_j will be fixed in the direction $A_i \to A_j$, thus A_i is put on the edge of the set of activities which have not yet been executed. For this way of branching, we can create a branching rule similar to the (SC) branching rule by choosing an activity which is closest to being put in front of the remaining activities by the edge finding rules and schedule it in as soon as possible.

We pick the critical machine by taking the machine of the activity from the consecutive cut (see Section 4.1.2) that has the lowest $r_j + p_j$ value. Then the activities which are candidates to be scheduled to process first are the activities on this critical machine for which all predecessors have been scheduled in. We determine how close an activity A_i is to being put in front of a set of activities Ω by calculating the JPS with the extra constraint that at least one activity $A_j \in \Omega$ is processed before A_i. If the score we get from calculating this JPS is close to the upperbound, we know that the A_i is also close to being put in front of the set of activities Ω by the edge-finding technique. We order the set of candidates from highest JPS-score to lowest JPS-score and start by planning in the activity with the highest JPS-score. We will refer to this branching scheme as *Plan Activity Edge Branching (PAEB).*

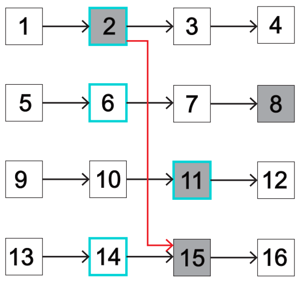


Figure 8.1: *Example where the consecutive cut consists of A_2, A_6, A_{11}, A_{14} (cyan). The ordering $A_2 \to A_{15}$ has already been chosen and the activities in grey belong to the machine for which we want to plan in the next activity. The head, body and tail of the activities are given in the Table 6.*

In the example of Figure 8.1 and Table 6, we want to plan in one of the grey activities from the consecutive cut: A_2 or A_{11}. For each activity A_i, we construct the JPS by scheduling at each moment the available activity with the longest tail, with the extra constraint that A_i cannot be executed first. For A_2, we process A_{11} for one time-unit, then we process A_8 until its finished, then we execute A_2 and A_{15} an then we process A_{11} for its remaining processing time, this gives a JPS-score of 20. For A_{11}, we process the activities in the order A_2, A_{15}, A_8, A_{11}, this gives a JPS-score of 15. Thus, we create a branch where we schedule A_2 as early as possible and if it turns out to be a dead-end, we create a branch where we schedule A_{11} as early as possible.

An advantage of (PAEB) is that there is always an activity which is closest to being propagated, thus we can always perform this heuristic (this is not the case for IEB, MDEB and JDEB). This branching scheme does not necessary result in a binary tree since there are in between 1 and $|J|$ possible children for each node.

A_i	A_2	A_8	A_{11}	A_{15}
r_i	3	8	7	5
p_i	2	3	2	3
q_i	7	3	2	4

Table 6: *Head, body and tail for the different gray activities of Figure 8.1.*

8.4 Amount of propagation

After choosing the ordering of a disjunction, we always propagate to try and get as much information from this one choice as possible. With this knowledge, we created another branching structure which counts the amount of information a disjunction will get by performing the propagation techniques (Edge-finding and Disjunctive Constraint Propagation) for each of the possible orderings on a critical machine and choosing the one which gives the most information. We call this branching scheme *Full Propagation Amount (FPA).*

Because performing disjunctive Constraint Propagation and edge-finding (twice) for each disjunction in every branch can get very costly, we tried a variation where we make a small selection of disjunctions and see how much fixing them would propagate. This selection consists of the ordering with the highest slack, and if they exist the two orderings which will cause forward and backward edge-finding to gain new information. We refer to this branching scheme as *Selection Propagation Amount (SPA).*

Another way of using the *amount of propagation* branching scheme is by only using it if most disjunctions have already been given a direction. We expect this branching scheme to be able to find branches that give lots of information, but in each node it costs $O(n\log(n)) * d)$ time, where n is the number of jobs and d is the number of disjunctions left to fix. Thus we can use a hybrid approach where, in the first nodes, we use one of the other heuristics and in the latter nodes (after a certain number of disjunctions have been given a direction), we use the amount of propagation for all of the remaining disjunctions. We call this hybrid approach *Hybrid Propagation Amount (HPA).*

9 Implementation

We parse a given instance of the JSP by creating a variable for each of the activities. These variables are all given a parameter for head, tail, size and *id*. The id of an activity A_i equals $nb\_of\_machines \times job\_number_i + machine\_number_i$. Here $nb\_of\_machines$ is the number of machines the problem has, $job\_number_i \in \{0, .., nb\_of\_jobs - 1\}$ is the job A_i belongs to. The size of each variable equals the processing time p_i of the corresponding activity. The head (tail) of the first (last) operation of each job is initialized as 0 and for every other activity, the head (tail) gets initialized as $r_i = r_{i-1} + p_{i-1}$ $(q_i = q_{i+1} + p_{i+1})$, where A_{i+1} is the successor of A_i in a job. We start with an empty list of chosen disjunctions and for every machine, create a list of disjunctions of the form (A_i, A_j) if $id_i < id_j$. (so we only have one tuple for each disjunction.) We start with an upper bound ub, we want to find a solution to the problem for which the solution-value $\leq ub$.

9.1 Upper bounds

To find an initial upper bound for the search, we perform simulated annealing. We find an initial solution for the simulated annealing by using the Florian, Trepant and Mcmahon scheme [16]. With this scheme, we only consider activities for which all of the predecessors have been planned in, thus we do not have to consider cycles in finding the initial schedule.

The neighbours that we use are obtained from a complete schedule by swapping two randomly chosen adjacent activities (belonging to the same machine) on a critical path of the complete schedule. The number of iterations in the simulated annealing is dependent on the size of a benchmark instance. We have chosen to use a number of iterations linear to n^2m^2 because the local search finds good initial solutions for small instances fast, but for bigger instances we want to give it some more time to find a better initial solution. Here n and m are the number of jobs and the number of machines in a benchmark instance respectively.

9.2 Lower bounds

After finding an initial upper bound with the simulated annealing, we start branching. In each node of the Branch-and-bound tree, we first solve a preemptive one-machine relaxation for each of the machines. In Branch-and-bound: if the relaxation of one of the machines has a score higher than the upper bound for which we try to find a schedule, we know that no schedule is possible and we backtrack. In Constraint Programming or if there is no such machine, then the machine with the highest relaxation becomes our critical machine and it serves as a lower bound for the current node. We prefer to find an ordering (or increase the head or tail of an activity) on this critical machine.

If the lower bound that we find is very close to the upper bound for which we try to find a solution, we find a better lower bound for the current choices by considering the non-preemptive one-machine subproblem. We solve this problem by using the implementation for the entire JSP but without considering the other machines. If the lower bound found with this subproblem is higher than the current upper bound, then we have found a dead end and we backtrack.

9.3 Branching

When we find a lower bound and a critical machine, we select an ordering to fix, a head or tail to increase or an activity to plan in by using one of the branching schemes of the previous section. Note that some of the branching schemes do not necessarily branch on a critical machine, in this case we just branch on a non-critical machine. Some branching schemes do not always find an ordering to branch on, if this happens we use the standard S-C branching scheme.

9.3.1 Selecting an ordering

When an ordering $(A_i \to A_j)$ is chosen, we perform a breadth-first-search in the partial schedule starting at activity A_j. We set $r_j = \max(r_i + p_i, r_j)$ and the successors of A_j are put in the queue. For each activity A_k we find with this BFS, we change the head $r_k = \max(r_{k-1} + p_{k-1}, r_k)$ and put its successors in the queue, here A_{k-1} is a predecessor of A_k. When we find A_i, we know that the ordering results in a cycle and we backtrack. For each of the machines, we keep a list of activities that we find while doing the BFS. Similarly, we perform a BFS in the reversed partial schedule starting at activity A_i. For each activity A_k we find, we change the tail $q_k = \max(q_{k+1} + p_{k+1}, q_k)$. For this reversed BFS, we also keep a list of all the activities we find for each of the machines. If after the BFS's, for one of the activities A_k we now have that $r_k + p_k + q_k > ub$, then we backtrack. After we perform both BFS's, we go over the two lists that we saved and for each of the machines, if an activity A_x is in the list of the normal BFS and another activity A_y is in the list of the reversed BFS, we know that we should fix the ordering between the two in the direction $A_y \to A_x$. Because we now fix all orderings $A_y \to A_x$ which would result in a cycle if given the opposite direction $A_x \to A_y$, this guarantees that we will not get cycles in our Disjunctive Graph.

9.3.2 Increasing head or tail

If we branch on increasing the head of an activity, then after we increase the head, we perform a BFS in the normal direction and for every activity A_k we find, we change the head to $r_k = \max(r_k, r_{k-1} + p_{k-1})$ where A_{k-1} is the predecessor of A_k in the BFS. When the head of an activity A_k does not change in an iteration of the BFS, then we do not put its successors in the queue since their head was already dependent on r_k. Different than selecting an ordering, because we are only increasing heads or tails, we can not have a cycle appear when branching in this manner, thus it is not necessary to perform a BFS on the entire partial graph. Similarly, if we increase the tail of an activity, we perform a BFS in the reversed direction and change the tails of the activities we find.

9.3.3 Scheduling an activity

We choose the machine on which we schedule an activity by finding the activity A_i from the consecutive cut for which $r_i + p_i$ is minimal (see Section 4.1.2). We then heuristically select an activity (A_c) from the consecutive cut to plan in and we fix every disjunction between A_c and other activities (for which no direction had been chosen) in the direction $A_c \to A_d$. We then perform a BFS starting at A_c where we increase the head of all activities that we find. The tail of A_c changes to $\max(q_c, \max_{(A_c,A_d)\in D} q_d + p_d)$.

9.4 Constraint Propagation

After we fix an ordering or increase the head/tail of an activity, we use the propagation rules (see below) to try and find more disjunctive arcs that have to get a direction (disjunctive Constraint Propagation), activities for which we can increase the heads and tails (edge finding) or dead-ends.

When propagating, we first perform Disjunctive Constraint Propagation and edge-finding for all of the machines, starting at the machine for which we fixed the most disjunctive arcs. While doing the first iteration of propagation, we keep track of how many changes each machine gets, both in added orderings or increases of heads or tails. If there are changes to at least one activity on a machine during an iteration of propagation, we perform the propagation techniques on this machines again. The order in which we perform the propagation techniques on each of the machines is from most changes to least changes. We stop propagating when none of the machines have any changes in an iteration. We implemented 4 orders of doing an iteration of propagation:

- First perform Edge-Finding for all of the machines, then perform Disjunctive Constraint Propagation for all of the machines.
- First perform Disjunctive Constraint Propagation for all of the machines, then perform Edge-Finding for all of the machines.
- Perform Disjunctive Constraint Propagation and then Edge-Finding for each of the machines.
- Perform Edge-Finding and then Disjunctive Constraint Propagation for each of the machines.

9.5 Backtracking

A node becomes a dead-end if all of its children are a dead-end, or if we know there is no schedule possible with the choices made so far; this can happen when a lower bound exceeds the upper bound, when the sum of the head, body and tail of an activity exceeds the upper bound, or when the Constraint Programming techniques propagate that there is no schedule possible. If we come across a dead-end and we have to backtrack, then one of six situations occurs:

- We were trying $A_i \to A_j$ in the first branch of a node. Now we go into the second branch of the node and we try $A_j \to A_i$
- We were trying $A_j \to A_i$ in the right branch of a node and we have already tried $A_i \to A_j$ (unsuccessfully). Now the entire node of giving a direction to the disjunctive arc (A_i, A_j) becomes a dead-end and we go back to its parent-node.
- We were trying $r'_j = r_j + x$ in the first branch of a node. We go into the second branch of the node and we try $q'_j = ub - p_j - r_j - x + 1$.
- We were trying $q'_j = ub - p_j - r_j - x + 1$ in the second branch of a node after we already tried $r'_j = r_j + x$. Now the node becomes a dead-end and we go back to its parent-node.
- We were trying to schedule A_i as early as possible, while having a list of activities Ω_i as alternatives. Now we take the next activity of Ω_i and try and schedule it as early as possible.
- We were trying to schedule A_i as early as possible, with $\Omega_i = \emptyset$. Now there are no alternatives left, so the node becomes a dead-end and we return to its parent-node.

If the root-node becomes a dead-end, then we know that there is no schedule possible for the current upper bound ub and we report that the optimal solution has value equal to at least $ub + 1$.

9.6 Leaves

If there are no disjunctions left to give a direction, we have found a leaf of the search tree. It should have a solution value $val \leq ub$ (otherwise we would have found a dead-end in an earlier node). Now we report that we have found a new upper bound to the problem, we lower ub to $val - 1$ and we go into *follow path mode*: we save every choice we made in the Branch-and-bound tree and when we find a leaf, we follow the choices that we made, try and perform them again and if we find a contradiction (from propagation or because the lower bound exceeds the upper bound) we make the opposite choice (or the next choice in the case of planning in an activity). After finding a contradiction, we go on and select branches in the same manner as before finding the leaf.

9.7 Branching strategies

Because we want to research the difference in performance of Constraint Programming, Branch-and-bound and Hybrid Constraint Programming and Branch-and-bound, we give an overview of the differences between the implementations of the three:

- In a Constraint Programming (CP) algorithm, we perform a tree search algorithm that uses the Constraint Propagation techniques Edge Finding and Disjunctive Constraint Propagation in each node of the search tree (Section 9.4). If we find new constraints, we add them to the current node and if we find a contradiction, we backtrack. For the algorithms that choose what machine is critical based on the non-preemptive one-machine lower bound, we still calculate the lower bound with the JPS, but we do not compare it to the known upper bound.

- In a Branch-and-bound (BB) algorithm, we perform a tree search algorithm that uses the JPS-bound as a lower bound for each node(Section 9.2). If this lower bound is higher than the current upper bound, we backtrack. If this lower bound is close to the current upper bound ($\geq 0.95 * ub$) and if we can get a higher lower bound by considering the non-preemptive one-machine problem, we calculate a new lower bound with the non-preemptive one-machine problem. If this new lower bound goes over the current upper bound, we backtrack.

- In a Hybrid algorithm (CP+BB), we perform the Constraint Propagation techniques (Section 9.4) and calculate a lower bound at every node of the search tree (Section 9.2). If either of the two finds a contradiction, we backtrack. We add the constraints that we find with the Constraint Propagation to the node.

In all three types of algorithms, we use local search to find an initial upper bound before traversing the search-tree (Section 9.1). We use the same upper bound for each of the algorithms to have a fairer comparison.

10 Experimental settings

We conduct the experiments on a computer with 16 GB RAM, windows 10 OS and a Ryzen 5 3600 processor. We implemented the different algorithms using Python 3.9.13.

The first experiment consists of testing all of the new branching schemes for each benchmark and comparing them with each other. For each benchmark instance, we first perform simulated annealing so that all of the branching schemes start with the same initial upper bound. For each test, we report the initial upper bound, the time it takes the algorithm to find the optimum solution and the number of branches it needs to find this optimal solution. If an algorithm does not find the optimum because of a timeout, we report that it timed out and report the best found solution so far.

For the second experiment, we use (a selection of) the newly created branching schemes (Section 8) and compare them with the branching scheme of Smith and Cheng [13] implemented as a Branch-and-bound algorithm (without Constraint Propagation) and as a Constraint Programming algorithm (without bounding).

In the third experiment, we test different algorithms based on the PAEB branching scheme. We test how well the PAEB works when we sort the candidates in the opposite order in each branch, and we test the performance of two variants of the PAEB algorithms: sorting the candidates from biggest to smallest tail q_i, and sorting the candidates from biggest to smallest $q_i - r_i$.

In the fourth experiment, we test different Edge-finding branching schemes, and test if it makes a difference if we take a different function to determine what combination of activity and set of activities we take. We test for combinations of "closest to edge" and "gaining the most". Where closest to edge has the highest $ub - \text{Score}_{c,\Omega}$ and with gaining the most is the combination for which, after we create a branch, the head of A_c will increase the most. We also test the influence of using Theorem 5 when using a branching scheme based on the Edge-finding score.

In the fifth experiment, we test if changing the propagation order has any influence on the performances of the algorithms.

10.1 benchmarks

The benchmarks that we use to test the different branching schemes on are: abz05, abz06 [1], ft06, ft10, ft20 [15], la01-la22, la24-la26,la28,la30,la35,la37,la40 [17] and orb01-orb10 [2].

11 Results and Discussion

For the initial upper bounds that we found with local search and the optimal solutions for each benchmark instance, see Appendix A. The results for JDEB in table 7 chooses the branch where the edge-finding propagation increases the head or tail of an activity the most and it does use the technique of Theorem 5. SC is the branching algorithm of Smith and Cheng [25] that chooses the ordering of a pair of activities that is closest to being imposed by the Disjunctive Constraint Propagation. "$\backslash P$" denotes that an algorithm does not use the Constraint Propagation techniques. "$\backslash B$" denotes that an algorithm does not calculate a lower bound in each node of the search tree. MDSB/JDSB, JDEB, FPA and PAEB are the algorithm described in Sections 8.1.2, 8.2.2, 8.4 and 8.3 respectively.

Table 7: *Results of the new BB + CP algorithms and comparison between BB, BB+CP and CP. For each benchmark instance, the time it takes the algorithm to find an optimal solution and prove that it is optimal is given in seconds (with decimals). If the algorithm does not find an optimal solution after 5 minutes, we report the best solution found so far (as a whole number). The best result for each instance is in bold (fastest time or best non-optimal solution). A '*' denotes that an algorithm was able to find an optimal solution, but did not prove its optimality within the time limit.*

Instance	SC	SC$\backslash P$	SC $\backslash B$	MDSB	JDSB	JDEB	FPA	PAEB	PAEB$\backslash P$	PAEB$\backslash B$
abz05	1263	-	1263	1242	1272	1262	**1236**	1268	1268	1268
abz06	24.67	-	23.35	**10.68**	105.66	29.82	53.50	-	-	-
ft06	0.017	1.960	0.023	**0.016**	0.021	0.027	0.051	0.031	0.487	0.031
ft10	**967**	-	**967**	987	1006	972	984	1061	-	1061
ft20	**1180**	1350	**1180**	1388	1388	1207	-	1410	-	1410
la01	0.126	-	0.127	0.089	0.078	0.125	0.505	**0.065**	-	**0.065**
la02	**0.629**	689	1.42	0.843	10.76	2.30	19.23	666	841	666
la03	3.92	653	**3.90**	3.94	9.22	15.81	17.28	625	-	625
la04	5.60	619	**5.56**	5.66	28.37	5.81	18.08	600	-	600
la05	0.232	**0.048**	0.250	0.085	0.109	0.273	0.247	0.065	0.096	0.065
la06	3.34	-	3.37	3.57	2.57	4.66	11.24	**0.221**	-	0.222
la07	**11.89**	970	11.92	53.50	934	23.20	921	915	-	915
la08	0.659	-	0.669	1.59	**0.631**	0.632	5.31	-	-	-
la09	5.837	0.602	5.92	2.68	24.80	5.50	5.69	**0.237**	-	0.238
la10	38.56	-	39.33	2.84	4.58	4.03	10.73	0.237	-	**0.221**
la11	6.45	-	6.38	38.86	**4.00**	9.54	39.07	-	-	-
la12	62.59	-	25.00	**23.90**	34.82	65.80	59.62	77.96	-	77.10
la13	51.95	-	52.84	**21.76**	-	180.67	49.67	-	-	-
la14	28.72	14.30	28.14	8.67	**8.05**	35.49	50.06	-	-	-
la15	**227.01**	1246	232.53	1212	1246	237.53	1251	-	-	-
la16	979	-	979	**964**	975	979	979	974	-	974
la17	5.20	-	5.12	15.97	66.75	**1.32**	26.68	792	-	792

Continuation of Table 7										
Instance	SC	SC$\backslash P$	SC $\backslash B$	MDSB	JDSB	JDEB	FPA	PAEB	PAEB$\backslash P$	PAEB$\backslash B$
la18	75.49	-	73.13	-	-	**32.68**	150.85	879	-	879
la19	181.83	-	176.81	**143.10**	842*	220.93	287.698	855	892	855
la20	916	-	916	907	942	**127.88**	909	929	-	929
la21	-	-	-	1131	-	**1116**	1130	-	-	-
la22	1004	1038	1004	1053	-	**962**	-	-	-	-
la24	1026	-	1026	-	-	**1025**	-	-	-	-
la25	1085	-	1085	1085	1085	**1039**	1091	-	-	-
la26	-	-	-	**1349**	-	-	-	-	-	-
la28	**1367**	-	**1367**	-	-	1416	-	-	-	-
la30	-	-	-	1536	**1533**	-	1535	-	-	-
la35	-	-	-	2042	2042	**2037**	-	-	-	-
la36	1370	-	1370	-	-	**1344**	-	-	-	-
la37	-	-	-	-	-	-	**1480**	-	-	-
la39	-	-	-	-	-	**1323**	-	-	-	-
la40	**1282**	-	**1282**	-	-	1318	1323	-	-	-
orb01	**1099**	1140	**1099**	1138	1199	**1099**	1195	1231	-	1231
orb02	925	-	925	889	**888***	925	889	940	-	940
orb03	**1087**	-	**1087**	1136	1149	1096	**1087**	-	-	-
orb04	192.518	1098	**185.989**	1094	1066	231.36	1078	1066	-	1066
orb05	906	-	906	952	972	**891**	-	938	-	938
orb06	**1071**	-	**1071**	1081	1093	1072	1096	1102	-	1102
orb07	409	-	409	**261.184**	416	404	397*	420	-	420
orb08	934	1032	934	959	1043	**50.37**	-	1042	-	1042
orb09	**939**	992	**939**	943	987	969	943	971	-	971
orb10	3.54	-	**3.31**	6.13	21.60	3.89	40.22	-	-	-

We see in Table 7 that the difference between Constraint Programming and Constraint Programming combined with Branch-and-bound (Column 1 and 3 or Column 8 and 10 in Table 7) is quite small. We expected the bounding to have a positive outcome on the time complexity of the algorithms, but in reality it seems to not matter much and it even slows the algorithms down in some cases (la02,la12,orb10 for SC). For PAEB, there seems to be less of an influence of using bounding in each node as the values of Column 8 and 10 are very close to each other for all of the benchmarks.

A possible reason for the influence of bounding being so small is that we use the JPS to find lower bounds for each branch while the edge-finding propagation is also calculated using the JPS. Edge-finding works by calculating the JPS for each subset of activities for a machine and it tries to find activities that must be executed at the border of one of these subsets. Naturally, we also consider the JPS for all of the activities of a machine and when this goes over the current upperbound, we have found a dead-end that is equal to the dead-end we would have found using the preemptive one-machine lower bound in the Branch-and-bound algorithms. For the Branch-and-bound algorithms, we also consider the full one-machine subproblem as a relaxation if the JPS gives a lower bound close to the upper bound. This might be the reason why Constraint Programming without Branch-and-bound is faster than Constraint Programming with Branch-and-bound for some of the instances. The full one-machine subproblem takes a significant amount of time and if it does not gain a lot of information (dead-ends), then it might slow the algorithm down.

Using the propagation has a very big effect on the speed of the algorithms. In all cases, except la05, la09 and la14 for SC, the SC algorithm with Constraint Programming was faster than SC without Constraint Programming or it was able to find solutions where SC\ P timed out after 5 minutes. SC without Constraint Programming was able to find an optimal solution for 4 out of 47 benchmark instances (ft06, la05, la09 and la14), while SC with Constraint Programming was able to an optimal solution for 22 out of 47 benchmark instances and in 26 out of 47 instances, SC was able to find a (sub-optimal) solution where SC\ P was not able to find a solution for its starting upper bound at all.

PAEB without Constraint Programming was able to find an optimal solution for 2 out of 47 instances, while PAEB with Constraint Programming was able to find an optimal solution for 7 out of 47 instances. The 2 instances for which PAEB\ P was able to find an optimal solution used 15.7 (ft06) and 1.5 (la05) times as many seconds as PAEB. In 22 out of 47 cases, PAEB was able to find a (sub-optimal) solution where PAEB\ P was not able to find any solution for its starting upper bound at all.

PAEB with Constraint Programming (with and without bounding combined) was the fastest algorithm for 4 out of 47 benchmark instances (la01, la06,la09 and la10) but in other instances (la02-04,la07,la08 la11-15), it was among the slowest of algorithms.

The algorithms MDSB and JDSB have some benchmarks where they are the fastest (abz06, ft06, la08, la11-14, la16, la19 and orb07) and MDSB was the only algorithm that found the optimal solution for orb07 within 5 minutes. JDEB works well for the instances la17,la18,la20-25, la26, la39, orb05 and orb08 where it was either the fastest algorithm that we tested, or it found the best (sub-optimal) solution out of all the algorithms.

Table 8: *For each benchmark instance, the number of branches the algorithms of table 7 needed to find an optimal solution is reported. If an algorithm did not find an optimal solution after* 5 *minutes, we report nothing. For each instance, the results of the algorithm that uses the lowest number of branches is in bold. Benchmark instances where no algorithm found an optimum are left out.*

Instance	SC	SC\\P	SC \\B	MDSB	JDSB	JDEB	FPA	PAEB	PAEB\\P	PAEB\\B
abz06	6550	-	6550	**1878**	22821	6547	2576	-	-	-
ft06	10	29571	10	**7**	16	11	8	39	5777	39
la01	55	-	57	25	**19**	48	20	51	-	51
la02	296	-	844	**202**	2487	1240	981	-	-	-
la03	1561	-	1563	1376	2806	5165	**1280**	-	-	-
la04	2475	-	2475	2437	15071	1503	**1041**	-	-	-
la05	125	207	127	34	26	126	**18**	51	366	51
la06	218	-	220	135	72	252	**60**	76	-	76
la07	**3158**	-	3160	4962	-	5377	-	-	-	-
la08	215	-	217	108	102	179	**35**	-	-	-
la09	539	927	541	62	2003	189	**29**	76	-	76
la10	3140	-	3142	95	85	212	**57**	76	-	76
la11	473	-	475	1494	143	417	**65**	-	-	-
la12	534	-	536	176	343	528	**68**	6385	-	6385
la13	516	-	518	173	-	2092	**66**	-	-	-
la14	459	740	461	162	207	487	**77**	-	-	-
la15	10972	-	10992	-	-	**8814**	-	-	-	-
la17	976	-	976	3905	15867	**227**	1501	-	-	-
la18	15420	-	15420	-	-	**8076**	8785	-	-	-
la19	40162	-	40162	30395	-	48196	**11882**	-	-	-
la20	-	-	-	-	-	**47641**	-	-	-	-
orb04	**38507**	-	**38507**	-	-	50134	-	-	-	-
orb07	-	-	-	**53540**	-	-	-	-	-	-
orb08	-	-	-	-	-	**7425**	-	-	-	-
orb10	**440**	-	**440**	715	3230	623	652	-	-	-

In Table 8, we see that FPA uses the least number of branches in almost all instances where it is able to find the optimal solution. However, if we look at the time it takes to find these solutions in Table 7, we see that it is slow compared to the other algorithms. This was to be expected because FPA uses a lot of computation to find the next branch in each node.

In the instances la05, la09 and la14, SC\P was able to find an optimal solutions while using less than double the number of branches of SC. These are also the only three instances where SC\P is faster than SC. In the instances la05, la09 and la14, SC find a total of 120, 1422 and 2512 disjunctive constraints respectively while doing the tree-search. It appears that in these three instances, the branching structure of Smith and Cheng is very good at finding the right branches to choose and even though the Disjunctive Constraint Propagation is able to find a lot of branches without having to make a choice, this is more costly than simply choosing the right branches. ft06 is the smallest benchmark instance that we tested on (6×6). The number of branches needed by SC\P to find an optimal solution is 2957 times as much as the number of branches needed by SC.

During the testing of the PAEB algorithm, we noticed that when we sort the candidates from lowest JPS-score to highest JPS-score, the algorithm would perform better for some of the benchmark-instances. Therefore, we included the results of the *Reversed PAEB* (R-PAEB) in Table 9. Also, because the activity with the highest JPS-score is often equal to the activity with the biggest tail, we tested the simple heuristic of sorting the candidates from biggest tail to smallest tail in each node. We refer to this branching algorithm as *Plan Activity Tail Branching* (PATB) and we also included it in Table 9. Finally, because the JPS-score of an activity with a relatively small head and a large tail will be relatively high, we implemented a similar heuristics that sorts the candidates from biggest to smallest $q_i - r_i$ value. We refer to this branching algorithm as *Plan Activity Head-Tail Branching* (PAHTB). The performances of the PAEB, R-PAEB, PATB and PAHTB algorithms are shown in Table 9.

instance	PAEB	R-PAEB	PATB	PAHTB
abz5	1268	1249	**1236**	1248
abz6	-	**967**	975	975
ft06	0.031	**0.026**	**0.026**	0.030
ft10	1061	**1025**	1046	1046
ft20	1410	1369	1190	**75.12**
la01	0.065	0.127	**0.059**	0.066
la02	666	**663**	666	666
la03	625	**10.61**	19.26	20.56
la04	600	**199.50**	590*	590*
la05	0.065	0.078	**0.055**	0.064
la06	0.221	0.253	**0.153**	0.169
la07	915	5.91	5.82	**4.30**
la08	-	**0.205**	0.208	0.218
la09	0.237	0.191	**0.163**	0.181
la10	0.237	0.222	**0.155**	0.171
la11	-	0.503	**0.367**	0.400
la12	77.96	0.472	**0.372**	0.413
la13	-	0.486	**0.366**	0.407
la14	-	0.441	**0.339**	0.377
la15	-	1250	11.47	**7.50**
la16	974	**973**	975	**973**
la17	792	**14.39**	792	792
la18	879	885	**854**	**854**
la19	855	**851**	864	854
la20	929	922	**915**	**915**
la21	-	-	1133	**1132**
la22	-	-	1032	**1025**
la24	-	1017	1025	**1004**
la25	-	-	**1062**	1073
la26	-	-	1350	**1347**
la28	-	1367	-	**1312**
la30	-	1473	1422	**1403**
la35	-	2026	1944	**183.22**
la36	-	-	-	**1341**
la37	-	-	-	**1465**
la39	-	-	-	**1323**
orb01	1231	1254	1266	**1224**
orb02	940	943	**917**	**917**
orb03	-	-	1169	**1150**
orb04	1066	1078	**1039**	1041
orb05	938	968	**932**	947
orb06	**1102**	-	-	-
orb07	420	-	**415**	**415**
orb08	1042	-	1014	**950**
orb09	971	**964**	986	986

Table 9: *Comparison between the PAEB, R-PAEB, PATB and PAHTB algorithms. Here decimal numbers denote the time it takes an algorithm to find an optimum and whole numbers denote the best score found after 5 minutes if an optimum was not found. If an algorithm was not able to find any solution after 5 minutes, we report nothing. For each instance, the best result is in bold. A '*' denotes that an algorithm was able to find an optimal solution, but not prove that it is optimal.*

Surprisingly, we see in Table 9 that using the reversed PAEB algorithm outperforms the normal PAEB algorithm in 26 out of 45 benchmark instances, they perform the same in 8 out of 45 instances and PAEB is faster or finds a better solution in the remaining 11 instances. A possible explanation why R-PAEB performs better than PAEB is that it is possible that in most branches, neither of the heuristics chooses the best activity to schedule in first. With R-PAEB, we choose the branch which results in the least *compact* schedules. By doing this, we follow a path in the search tree for which the makespan of different machines are getting close to the upper bound much faster, thus we either get new information from the Constraint Propagation, or we find a dead-end. Suppose there are three choices of activities A_1, A_2 and A_3 to schedule in before the others and they are ordered by some heuristic. Then, if in an optimal solution, A_2 is processed first;

if we first explore the branch where we choose A_3, we might get to the branch where we choose A_2 sooner than if we would have chosen A_1 first because the branch with A_3 results in a dead-end sooner. Now, the algorithm still has to prove the optimality of the solution by traversing the rest of the search-tree, but because the less-compact choice of R-PAEB leads to a new upper bound sooner than the compact PAEB, we get to use the new upper bound in the rest of the search-tree. This argument is strengthened by the following: for all of the benchmark instances where we get big differences between the performance of PAEB and R-PAEB (la03, la07, la08, la11, la12, la13, la14 and la17), starting a search from the value of an optimal solution minus 1 will result in a dead-end very fast (see Table 10).

Algorithm		la03	la07	la08	la11	la12	la13	la14	la17
PAEB	C	0.181	1.28	-	-	-	-	-	13.78
	$C-1$	0.003	0.001	0.003	0.001	0.002	0.000	0.002	10.56
R-PAEB	C	0.079	0.290	0.266	0.606	0.581	0.608	0.538	10.725
	$C-1$	0.003	0.001	0.002	0.000	0.002	0.001	0.002	10.38

Table 10: *Comparison between an initial upper bound of C (optimum) and $C-1$ for the different benchmarks where PAEB and R-PAEB perform unexpected. Here C is the known optimum (see Table 13 in Appendix A).*

Not only do the algorithms R-PAEB, PATB and PAHTB outperform the PAEB algorithm, they also perform very well compared to the other algorithms of Table 7. In 22 out of 47 instances (abz05, ft20, la01, la05-15 la24, la26, la28, la30, la35-37 and la39), PAHTB outperforms all of the algorithms of Table 7 and PAHTB was the only algorithm of the once that we tested which found an optimal solution for the instances ft20 and la35. For other instances (abz6, la02, la03, la04, la18, la19 and most of the orb01-10 instances) PAEB, R-PAEB, PATB and PAHTB were among the worst performing algorithms of the once that we tested. A possible reason why this PAHTB algorithm (combined with Constraint Programming) works so well compared to the other algorithms is that the PAEB algorithm (and variants) and the Constraint Programming techniques are able to omit parts of the search space (see Section 4.1.2). Different than Constraint Programming and Branch-and-bound, the parts of the search-space that we omit with the PAEB are not found by calculating the JPS.

In Table 11, we compare the variant of JDEB that chooses the combination of an activity and a set of activities with the highest score, with the variant of JDEB that branches on the activity for which the head or tail of the activity increases the most after branching. And we compare the JDEB that uses Theorem 5 with JDEB which does not use Theorem 5.

Instance		Score + Thm.5	Increase + Thm.5	Score	Increase
abz06	time(s)	27.05	29.82	944	**14.68**
	branches	6459	6547	-	2773
ft06	time(s)	0.033	0.027	**0.022**	0.037
	branches	20	11	9	31
la01	time(s)	0.159	**0.125**	0.335	0.154
	branches	37	48	103	43
la02	time(s)	**1.89**	2.30	9.74	2.29
	branches	922	1240	5788	1335
la03	time(s)	22.59	15.81	68.01	**3.47**
	branches	8467	5165	25911	1174
la04	time(s)	**2.44**	5.81	31.45	4.82
	branches	829	2503	13932	1953
la05	time(s)	0.275	0.273	0.302	**0.141**
	branches	124	126	53	62
la06	time(s)	4.84	4.66	**3.84**	3.95
	branches	245	252	299	236
la07	time(s)	25.24	23.20	966	**12.30**
	branches	4965	5377	-	2727
la08	time(s)	0.645	**0.632**	7.44	8.07
	branches	165	179	986	355
la09	time(s)	5.60	**5.50**	7.79	7.52
	branches	226	189	214	483
la10	time(s)	3.99	4.03	**3.50**	3.71
	branches	213	212	149	172
la11	time(s)	22.58	**9.54**	17.65	-
	branches	402	417	267	-
la12	time(s)	65.82	65.80	63.00	**62.22**
	branches	528	528	533	540
la13	time(s)	**39.52**	180.67	52.62	-
	branches	320	2092	427	-
la14	time(s)	44.41	**35.49**	42.20	46.45
	branches	487	487	408	607
la17	time(s)	3.40	**1.32**	22.19	33.20
	branches	622	227	4816	7960
la18	time(s)	31.12	32.68	878	**15.69**
	branches	8097	8076	-	3206
la19	time(s)	238.62	220.93	846	**172.02**
	branches	55010	48196	-	31158
orb10	time(s)	5.68	**3.89**	944*	34.61
	branches	928	623	-	3736

Table 11: *Results of comparison between using edge-score to choose a branch and using the number of units the head or tail of an activity increases to choose a branch. And the comparison between using and not using Theorem 5. The time values with decimals denote the time it takes the algorithm to find an optimal solution in seconds. The whole numbers denote the best solution found after the time out of 5 minutes. The best result for each instance is in bold. A '*' denotes that an algorithm was able to find an optimum but not prove its optimality.*

In 7 out of 20 cases, not using Theorem 5 and picking the combination of activity and set of activities for which we gain the most increase in head or tail is the fastest. In 7 out of 20 cases using Theorem 5 and branching on the amount of increase in head or tail is the fastest. Branching on the amount of increase has a little advantage over branching on the edge-score, but this advantage seems to be very dependant on the benchmark that we test the different branching rules on. If we look at instance la04, we see that using Theorem 5 has a very positive effect if we branch on the edge-score (2.44 seconds and 31.45 seconds), but if we branch on how much a head or tail increases, we see a (small) negative effect in using Theorem 5 (5.81 seconds versus 4.82 seconds), so apparently the combination of branching based on edge-score and using Theorem 5 is what makes this scheme faster for this instance.

In Table 12, we compare the different orders of executing the propagation rules. We have:

- *de+lw*: for each machine, first perform Disjunctive Constraint Propagation until it stops finding new constraints, then for each machine perform Edge-finding Propagation until it finds no new constraints.
- *ed + lw*: for each machine, first perform Edge-finding Propagation until it finds no new constraints, then for each machine perform the Disjunctive Constraint Propagation until it find no new constraint.
- *de+lo*: for each machine, perform Disjunctive Constraint Propagation followed by Edge-finding until no new constraints are found.
- *ed+lo*: for each machine, perform Edge-finding followed by Disjunctive Constraint Propagation until no new Constraint are found.

For the tests of Table 12, we use the SC algorithm that uses both bounding and propagation.

instance	de+lw	ed+lw	de+lo	ed+lo
abz6	23.64	**21.24**	26.11	26.44
ft06	0.017	**0.016**	0.020	0.019
la01	0.125	**0.119**	0.130	0.120
la02	1.46	1.38	1.48	**1.30**
la03	3.71	**3.47**	3.98	3.84
la04	5.90	**5.64**	6.04	5.90
la05	0.253	0.243	0.247	**0.242**
la06	3.47	3.44	3.45	**3.31**
la07	9.00	8.98	9.10	**8.70**
la08	0.672	**0.630**	0.650	0.634
la09	6.02	**5.85**	5.97	5.72
la10	39.99	**35.77**	38.33	36.49
la11	6.44	6.32	6.36	**6.21**
la12	63.89	62.57	62.62	**60.44**
la13	53.37	54.49	53.02	**52.29**
la14	28.64	28.54	28.90	**27.75**
la17	5.29	**4.91**	5.56	5.79
la18	76.10	**60.92**	78.21	81.03
la19	182.63	**156.68**	190.16	200.90
orb04	182.14	**154.23**	186.00	199.25
orb10	3.49	**3.24**	3.64	3.68

Table 12: *Time in seconds it takes the SC algorithm for each benchmark for different orders of propagation. The best result for each instance is in bold.*

In 20 out of 21 instance *ed+lw* is faster than *de+lw*. In 15 our of 21 instances, *ed+lo* is faster than *de+lo*. In 13 out of 21 instances, *de+lw* is a than *de+lo*. In 12 out of 21 instances, *ed+lw* is than *ed+lo*. The differences within each benchmarks are not very significant, but the fact that *ed* performs better than *de* in more than 71% for both *lw* and *lo* implies that performing Edge-finding and then Disjunctive Constraint Propagation is a little faster than doing it the other way around.

12 Conclusions

In this thesis, we set out to find what the effects of combining Branch-and-bound and Constraint Programming are for the Job Shop Problem. We started by finding out what techniques are used in the literature and used these as building blocks for our own algorithms. We found different Constraint Programming techniques in the literature and used the Constraint Programming rules to create new branching heuristics. By using branching rules which find orderings or activities that are close to being imposed by the propagation rules, we created algorithms whose goal is to gain the most information out of each branch and thereby reducing the size of the search tree. We then implemented the different algorithms and compared their performance on different benchmarks instances.

The goal was to find algorithms which reduce the speed of traversing a search-tree for the JSP by reducing its size. For most benchmarks and for most algorithms, reducing the size of the search-tree seemed to indeed speed up the algorithms. For the FPA algorithm, which uses Constraint Propagation for each possible choice of orderings between two pairs, we saw that we get relatively small search trees, but this algorithm is one of the slowest among the ones that we tested in most benchmarks. This tells us that reducing the search-tree is a good thing, but using too many calculations to reduce the search-tree might be more costly than the amount of time we gain by making it smaller. We see the same thing in benchmarks la05, la09 and la14, where the SC algorithm uses 125, 539 and 459 branches with propagation and 207, 927 and 740 branches without propagation. These are the only three cases where using Constraint Programming is not beneficial because in these cases, the calculation costs for the Constraint Propagation outweigh the gains we get for reducing the search tree.

We found that adding Constraint Propagation to a Branch-and-bound algorithm speeds up the algorithm by a lot for both SC and PAEB and our recommendation would be that any exact method for the JSP should contain at least some of these Constraint Propagation techniques. Adding the bounding after each node to a Constraint Programming approach does not give much of an advantage to the Constraint Programming algorithms since this is already being done by the edge-finding propagation and we would recommend to let the bounding be done by the different Constraint Propagation techniques. The Constraint Propagation techniques give bounds that are lower than the bounds for the (full) one-machine subproblem, but they are faster to compute and by doing the Constraint Propagation, we get these bounds “for free”: they require no extra computation.

We saw that different algorithms work well on different benchmarks and that these differences can be quite big. Where PAHTB was the fastest algorithm in some instances, it would be among the slowest for other instances. The three different algorithms SC, JDEB and PAHTB all have different instances where they outperform the others and together they use all three different kinds of branching (fixing ordering, planning activity or splitting start-interval), thus we think that a combination of these different algorithms might improve the performances.

We also tested the impact that Theorem 5 has on the performance of the JDEB algorithm. We saw that it does change the time it takes the JDEB algorithm to find optima, but it does not improve the performance of the algorithm in all benchmarks. Branching on how much a head or tail of an activity increases after the edge-finding imposed by increasing the head or tail of some activity seems to work better than branching based on how close an activity is to being put on the edge of a set of activities by the edge-finding propagation(14 out of 20 instances) but not in all instances.

12.1 Future work

There are still some improvements that can be made for the implementation of the solver. Now, we only consider Edge-finding and Disjunctive Constraint Propagation, but this could be extended with the other propagation rules from Section 5. The new branching structures that were created are also based only on Disjunctive Constraint Propagation and Edge-finding, so considering the other propagation rules might result in new branching structures. We could optimize some of the code that was written, for instance, we could use the information that we get from a dead-end to determine what combination of constraints is causing the dead-end and use this information in different branches.

If the window of executing of an activity A_i is smaller than $2p_i$, then we know A_i should execute in the interval $[ub - q_i - p_i, r_i + p_i]$ and its machine is thus occupied in this time window. In the current implementation, we only considered the arc-B-consistency (Section 4) of the activities, thus by creating these holes in the windows of execution of the intervals, we might be able to gain some information.

In the tests we did in this thesis, we limited the time to *5* minutes for each combination of algorithm and benchmark instance because there were a lot of algorithms to test on a lot of benchmark instances to test them on. We would like to increase this time and test for other (bigger) benchmark instances so that we have more data points to compare the algorithms on. Especially the bigger instances which require multiple hours to solve might give more inside on what the different algorithms are good at.

The different algorithms performed very different by the different benchmarks. Thus to always use the same algorithm would not be optimal. If we could estimate what algorithm would work best on given benchmark before having to run the entire instance, we could then make a choice on what algorithm we would want to use. Some research could be done to classify benchmark instances before using the algorithms on them. For instance, we could create a measure of how much an instance is like a flow shop problem or how much the different machines are alike. One can imagine that the PAEB algorithm works very well if the consecutive cut contains activities for all of the different machines since it makes the number of options in a node smaller.

References

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

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

[3] Baptiste, P., and Le Pape, C. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *IJCAI (1)* (1995), Citeseer, pp. 600–606.

[4] Baptiste, P., Le Pape, C., and Nuijten, W. *Constraint-based scheduling: applying constraint programming to scheduling problems*, vol. 39. Springer Science & Business Media, 2001.

[5] Barker, J. R., and McMahon, G. B. Scheduling the general job-shop. *Management Science 31*, 5 (1985), 594–598.

[6] Beck, J. C. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research 29* (2007), 49–77.

[7] Beck, J. C., Feng, T., and Watson, J.-P. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing 23*, 1 (2011), 1–14.

[8] Brucker, P., Jurisch, B., and Sievers, B. A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics 49*, 1-3 (1994), 107–127.

[9] Carlier, J., and Pinson, É. An algorithm for solving the job-shop problem. *Management science 35*, 2 (1989), 164–176.

[10] Carlier, J., and Pinson, E. A practical use of jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research 26*, 1 (1990), 269–287.

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

[12] Caseau, Y., and Laburthe, F. Improved clp scheduling with task intervals. In *ICLP* (1994), Citeseer, pp. 369–383.

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

[14] Colombani, Y. Constraint programming: an efficient and practical approach to solving the job-shop problem. In *International Conference on Principles and Practice of Constraint Programming* (1996), Springer, pp. 149–163.

[15] Fisher, H. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling* (1963), 225–251.

[16] Florian, M., Trepant, P., and McMahon, G. An implicit enumeration algorithm for the machine sequencing problem. *Management Science 17*, 12 (1971), B–782.

[17] Lawrence, S. Resouce constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University* (1984).

[18] Martin, P., and Shmoys, D. B. A new approach to computing optimal schedules for the job-shop scheduling problem. In *International Conference on Integer Programming and Combinatorial Optimization* (1996), Springer, pp. 389–403.

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

[20] Nowicki, E., and Smutnicki, C. Some new ideas in ts for job shop scheduling. In *Metaheuristic Optimization via Memory and Evolution*. Springer, 2005, pp. 165–190.

[21] Nuijten, W. P., and Aarts, E. H. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research 90*, 2 (1996), 269–284.

[22] Pardalos, P. M., Shylo, O. V., and Vazacopoulos, A. Solving job shop scheduling problems utilizing the properties of backbone and "big valley". *Computational Optimization and Applications 47*, 1 (2010), 61–76.

[23] Philipoom, P. R., and Fry, T. D. The robustness of selected job-shop dispatching rules with respect to load balance and work-flow structure. *Journal of the Operational Research Society 41*, 10 (1990), 897–906.

[24] Plotkin, S. A., Shmoys, D. B., and Tardos, É. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research 20*, 2 (1995), 257–301.

[25] Smith, S. F., and Cheng, C.-C. Slack-based heuristics for constraint satisfaction scheduling. In *AAAI* (1993), pp. 139–144.

[26] Sotskov, Y. N., Tautenhahn, T., and Werner, F. On the application of insertion techniques for job shop problems with setup times. *RAIRO-Operations Research-Recherche Opérationnelle 33*, 2 (1999), 209–245.

[27] Sourd, F., and Nuijten, W. Multiple-machine lower bounds for shop-scheduling problems. *INFORMS Journal on Computing 12*, 4 (2000), 341–352.

[28] Teppan, E., and Da Col, G. Automatic generation of dispatching rules for large job shops by means of genetic algorithms. In *CIMA@ ICTAI* (2018), pp. 43–57.

[29] van Hoorn, J. Jobshop instances and solutions. http://jobshop.jjvh.nl/. Accessed on: 2022-07-16.

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

[31] Vandevelde, A., Hoogeveen, H., Hurkens, C., and Lenstra, J. K. 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 (2005), 305–320.

[32] Werner, F., and Winkler, A. Insertion techniques for the heuristic solution of the job shop problem. *Discrete applied mathematics 58*, 2 (1995), 191–211.

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

Appendix A: Initial upper bounds and optima.

Table 13: *Initial upper bound and optimal solutions for all of the benchmark instances that we used in Section 11. The optima where obtained from [29].*

instance	init. ub	optimum
abz05	1276	1234
abz06	976	943
ft06	55	55
ft10	1074	930
ft20	1410	1165
la01	666	666
la02	977	655
la03	653	597
la04	644	590
la05	593	593
la06	926	926
la07	985	890
la08	863	863
la09	951	951
la10	958	958
la11	1222	1222
la12	1039	1039
la13	1150	1150
la14	1292	1292
la15	1251	1207
la16	979	945
la17	795	784
la18	891	848
la19	893	842
la20	953	902
la21	1136	1046
la22	1055	927
la24	1027	935
la25	1092	977
la26	1350	1218
la28	1417	1216
la30	1536	1355
la35	2042	1888
la36	1371	1268
la37	1480	1397
la39	1323	1233
la40	1324	1222
orb01	1270	1059
orb02	945	888
orb03	1170	1005
orb04	1099	1005
orb05	981	887
orb06	1102	1010
orb07	429	397
orb08	1048	899
orb09	1012	934
orb10	944	944