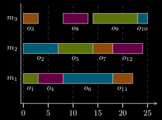
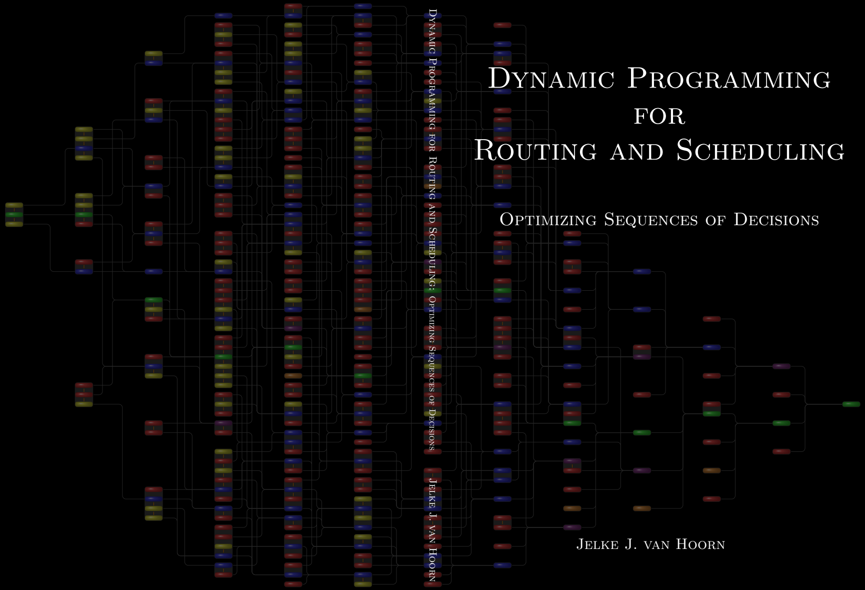
Dynamic Programming for Routing and Scheduling

Optimizing Sequences of Decisions

Jelke J. van Hoorn

Dynamic Programming for Routing and Scheduling: Optimizing Sequences of Decisions
Jelke J. van Hoorn





Job 1

	$m(o)$	$p(o)$
o_1	1	3
o_5	2	7
o_9	3	9

Job 2

	$m(o)$	$p(o)$
o_2	2	7
o_6	1	10
o_{10}	3	2

Job 3

	$m(o)$	$p(o)$
o_3	3	3
o_7	2	4
o_{11}	1	4

Job 4

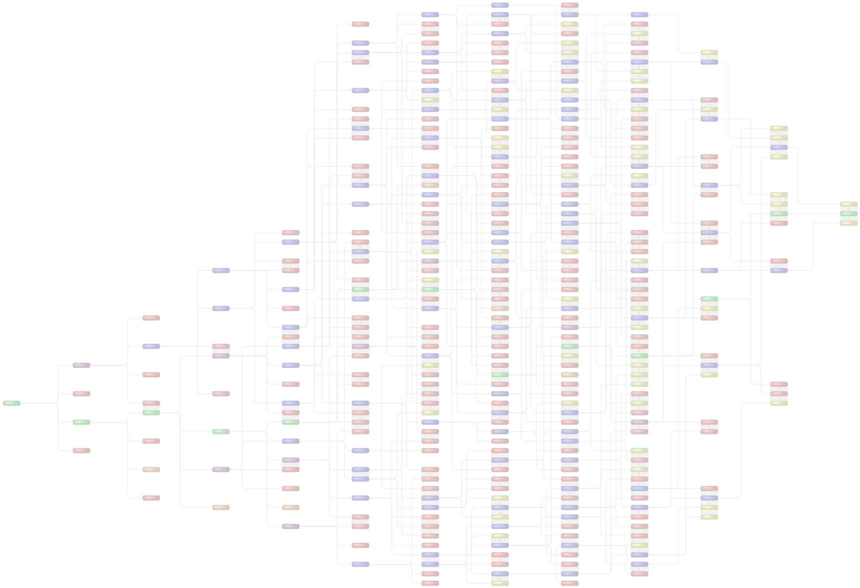
	$m(o)$	$p(o)$
o_4	1	5
o_8	3	5
o_{12}	2	6

JSSP

C_o	**Finish time of operation o**
j	**Job**
m	**Machine**
$C_{\max}$	**Makespan**
$p_{\max}$	**Maximum operation time**
o	**Operation**
$\pi_j(i)$	**i-th machine job j has to visit**
p_o	**Processing time of operation o**
ψ	**Schedule**
$\alpha(\varsigma, o)$	**Aptitude**
$j(o)$	**Job for operation o**
$m(o)$	**Machine for operation o**
$\lambda(S)$	**Last operation in S for each job**
$\varepsilon(S)$	**Next operation per job not in S**
$p(o)$	**Processing time for operation o**
$\eta(\varsigma)$	**Possible expansions of ς**
$\Lambda(\varsigma)$	**Last operation in the sequence ς**
N	**Number of jobs**
M	**Number of machines**
$\mathcal{J}$	**Set of jobs**
$\mathcal{M}$	**Set of machines**
$\mathcal{O}$	**Set of operations**
$\vec{\alpha}$	**Array of aptitude values**
$\vec{\eta}$	**Array of possible expansions**

JSSP Bounding

o^*	**Current operation**
$h^{\max}$	**First start of next maintenance**
$h^{\min}$	**First end of prev maintenance**
r_o	**Head of operation o**
$\tilde{r}_o$	**temporary head of operation o**
p_o^+	**Remaining processing time**
q_o	**Tail of operation o**
K^*	**Maximal set creating a block**
t	**Current time**
$t^{\mathbf{req}}$	**Next relevant time**
$\mathcal{A}$	**Set of available operations**
$\mathcal{D}$	**Set of delayed operations**
$\mathcal{U}$	**Set of unavailable operations**
$\mathcal{M}^+$	**Machines with operations left**
$\bar{I}$	**Set of all operations**
I	**Set of job operations**
$\breve{I}$	**Set of maintenance operations**



DP

β	**Bookkeeping variables**
E	**Number of expansions per solution**
H	**Number of expanded solutions**
$\doteq$	**Domination: equal**
$\gtrdot$	**Domination: dominates**
$\not\doteq$	**Domination: not comparable**
$\diamond\!\Rightarrow$	**Expands solution with node**
γ	**Comparable variables**
ϕ	**Fixed variables**
$\prec$	**Precedence relation between nodes**
Ω	**Set identifiers of optimal solutions**
ς	**A Solution**
$\mathring{\varsigma}$	**An optimal solution**
$\wr$	**Splits ϕ and γ in a state definition**
$\wr$	**Splits γ and β in a state definition**
ξ	**State**
$\breve{\xi}$	**Optimal solution**
$\hat{\xi}$	**Non-dominated solutions**

TSP

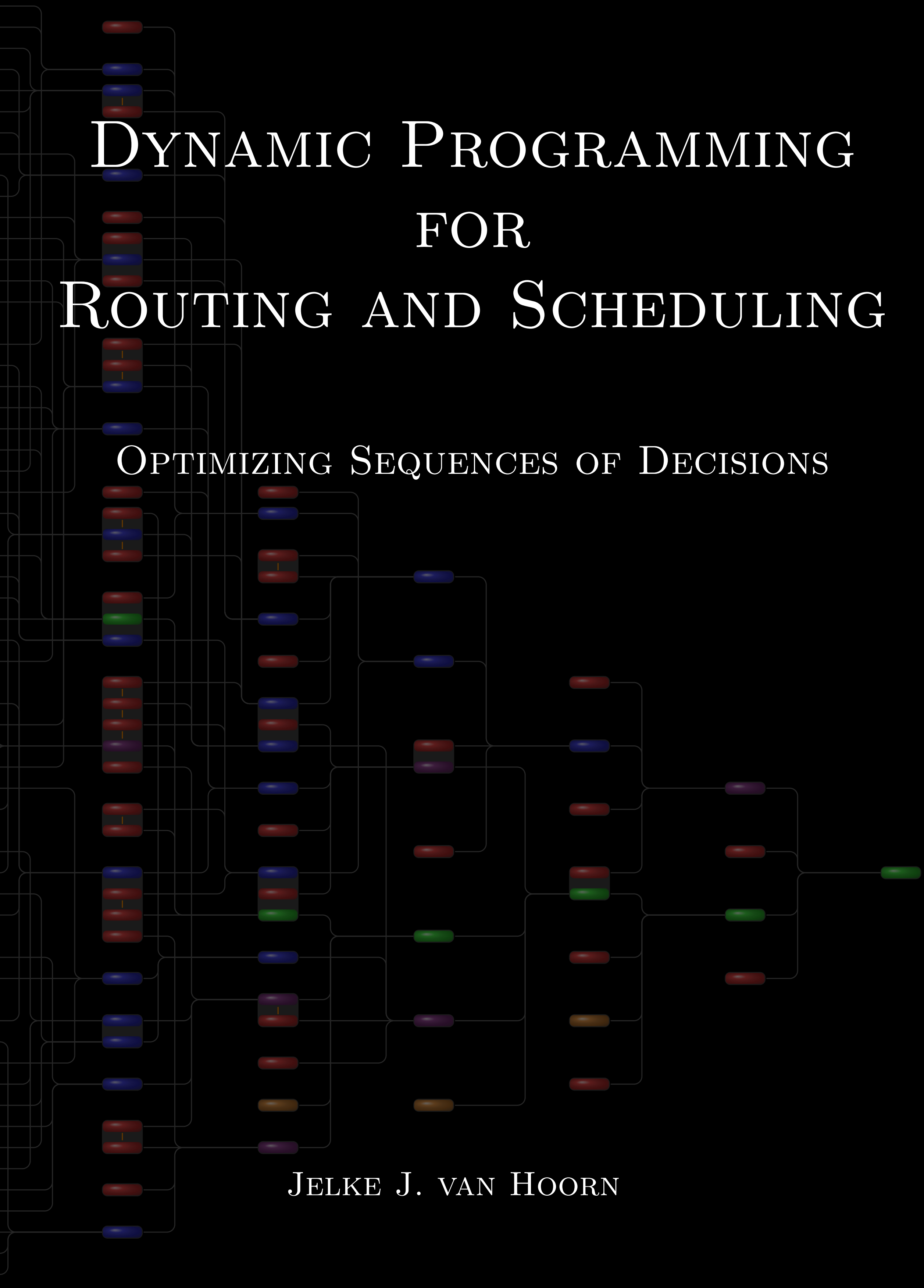
n	**Number of nodes of a TSP problem**
s	**Start node of a TSP used in DP**

VRP

d	**Destination of a vehicle**
o	**Origin of a vehicle**
r	**Request**
v	**Vehicle**
n	**Number of customer requests**
m	**Number of vehicles**
D	**Set of destinations**
O	**Set of origins**
R	**Set of requests**
V	**Set of vehicles**

JSSPM

D	**Downtime**
$\vec{u}$	**Array of left uptime**
u	**Left uptime**
R	**Maintenance**
U	**Uptime**
$\mathcal{N}$	**Number of maintenances**
$\mathcal{R}$	**Set of maintenances**



Dynamic Programming for Routing and Scheduling

Optimizing Sequences of Decisions

Jelke J. van Hoorn

Dynamic Programming for Routing and Scheduling

Optimizing Sequences of Decisions

Jelke J. van Hoorn

ISBN 978-94-6332-008-5

An electronic version of this dissertation is available at
http://dare.ubvu.vu.nl/ and http://jobshop.jjvh.nl/dissertation

VRIJE UNIVERSITEIT

Dynamic Programming for Routing and Scheduling

Optimizing Sequences of Decisions

ACADEMISCH PROEFSCHRIFT

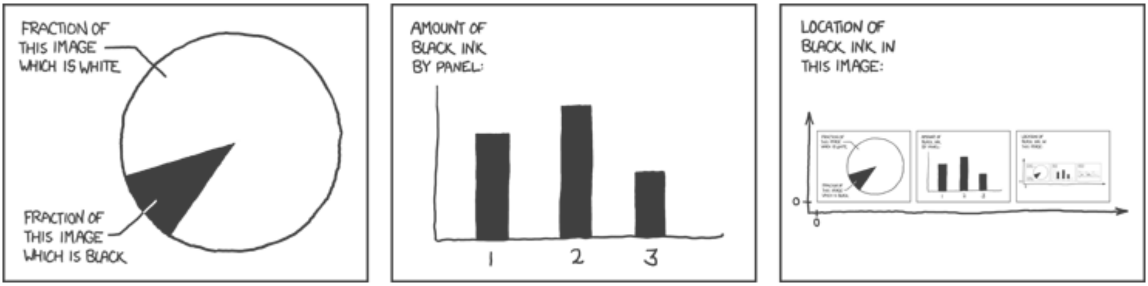
ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Economische Wetenschappen en Bedrijfskunde
op vrijdag 10 juni 2016 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

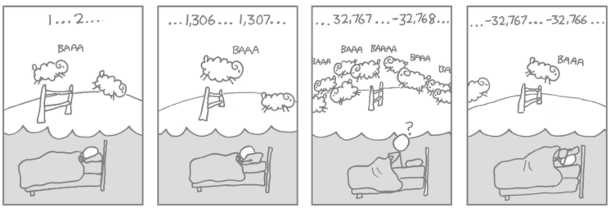
Jelke Jeroen van Hoorn

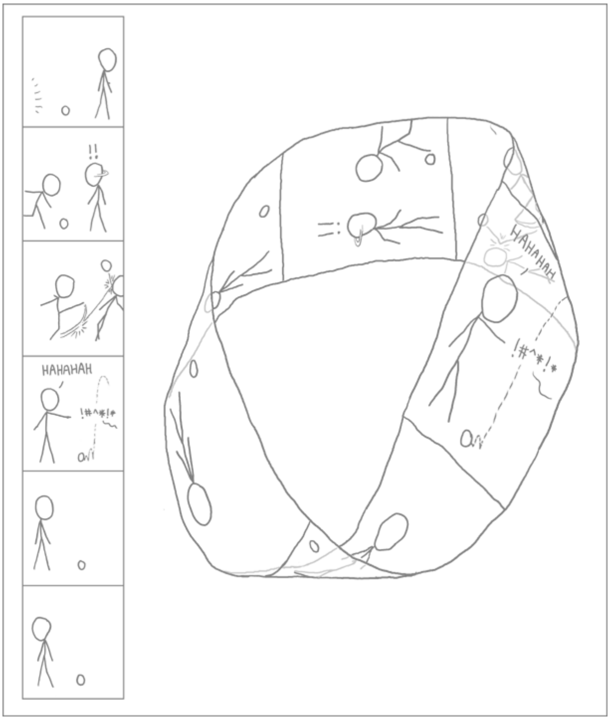
geboren te Delft

promotoren: prof.dr. J.A.S. Gromicho
prof.dr. G.T. Timmer

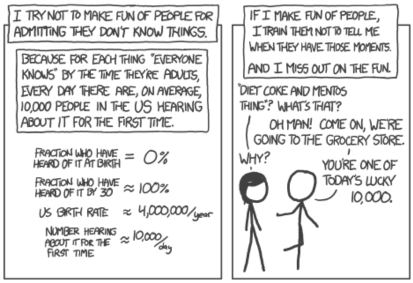

FRACTION OF THIS IMAGE WHICH IS WHITE
FRACTION OF THIS IMAGE WHICH IS BLACK
AMOUNT OF BLACK INK BY PANEL:
1
2
3
LOCATION OF BLACK INK IN THIS IMAGE:
0
0

Contents


1...2...
BAAA
...1,306... 1,307...
BAAA
...32,767...-32,768...
BAAA BAAAA BAAA
?
...-32,767...-32,766...
BAAA



!!
HAHAHAH
!#^*!*
HAHAHAH
!#^*!*



Introduction

This thesis is about Dynamic Programming and in particular about algorithms based on the algorithm designed by Held and Karp (and also independently by Bellman) offering the best complexity known today to optimally solve the Traveling Salesman Problem.

We see this algorithm as a general way to solve problems for which each solution can be represented as a sequence of nodes and where the goal can be defined as finding the best sequence. If all sequences have to be evaluated then a brute-force approach would result in an $\mathcal{O}(n!)$ algorithm as there are $n!$ ways to create a sequence of n unique nodes. With Dynamic Programming this effort can be reduced as Dynamic Programming evaluates in principle one partial sequence per subset. Since there are $\mathcal{O}(2^n)$ subsets, Dynamic Programming can exponentially decrease the effort of evaluating all sequences compared to the full enumeration of all sequences.

Such a Dynamic Programming algorithm has mainly theoretical implications as an algorithm providing the best known time complexity guaranteed to solve a problem to optimality. Since all possible sequences are stil evaluated implicitly, the time complexity of such an algorithm is still exponential, although largely exponentially ($\frac{n!}{2^n} \approx \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$) less then explicit evaluation of all sequences. Since multiple partial sequences have to be kept in memory to be evaluated later, the memory requirement of such an algorithm is also exponential. Both the exponential run time and the exponential memory requirements make practical use of such an algorithm limited. However, parts of the state space may be evaluated implicitly by adding bounding arguments, similar to branch and bound. Also converting a Dynamic Programming algorithm into a heuristic raises its practical value. With proper bounding good solutions can be found by using only a very small part of the state space. Sometimes bounding can even be so restrictive that the complete state space can be evaluated in reasonable time.

This dissertation emerged while researching the applicability of general optimization frameworks for vehicle routing at ORTEC. Dynamic Programming proved to be a very rich modeling framework for routing, handling restrictions which are known to challenge other techniques. Scheduling lacked such a rich modeling framework, for instance adding maintenances challenges the integer linear models beyond usability. Applying Dynamic Programming to the Job Shop Scheduling Problem was therefore a goal, with the added benefit of bringing a new rich modeling framework to scheduling, which we could use to solve the challenging version with maintenances.

We believe that contributing with a new optimal algorithm for the Job Shop Scheduling Problem helps reviving the combinatorial optimization research and shows that Dynamic Programming is still a powerful technique from both the theoretical and the practical points of view.

The first chapter, *Dynamic Programming*, gives an introduction to Dynamic Programming in general and how to use Dynamic Programming to find solutions for problems that can be represented as sequences of nodes by finding optimal solutions for subsets. The problems described in the chapter *Dynamic Programming* are solely for illustrative reasons, we show how to solve for instance the bipartite matching problem by an algorithm that requires an exponential effort. This is clearly not a good idea neither for theoretical nor for practical reasons, since the problem can be optimally solved by efficient deterministic polynomial algorithms. However, it serves important didactic purposes in understanding the rest of this thesis.

While the first chapter does not aim at providing an algorithm with the best known complexity to solve a problem to optimality, the second chapter does. This chapter, *Sequencing, Routing and Scheduling*, describes Dynamic Programming algorithms for the well-known Traveling Salesman Problem, Vehicle Routing Problem and Job Shop Scheduling Problem. The first, is the well-known algorithm of Held and Karp [62] and Bellman [17]. The second, and third are novel and appeared in [59,60]. Dynamic Programming provides the Job Shop Scheduling Problem with the first non Brute-Force optimal algorithm known to us which is not based on Branch and Bound. It also has the best known complexity to solve the problem to optimality.

Ideas similar to Branch and Bound can be used to improve the performance of a Dynamic Programming algorithm. This is described in the third chapter, *The Dynamic Programming State Space*. Also a procedure to find all optimal solutions as well as ways to convert an optimal Dynamic Programming algorithm into a heuristic can be found in this chapter.

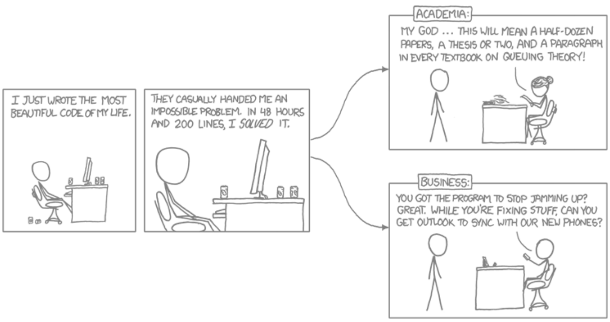
Chapter four, *The Vehicle Routing Problem*, shows the effects of bounding on the Dynamic Programming algorithm for the Vehicle Routing Problem. For this we use computational results on well-known benchmark instances for the Capacitated Vehicle Routing Problem. Furthermore, it contains an overview of several extensions of the Vehicle Routing Problem. We show how these extensions can be solved by extending the Dynamic Programming algorithm. This creates a general modeling framework able to tackle most of the challenging versions of Vehicle Routing Problem found in the literature, including some that have received almost no attention so far. These are extensions of our work which appeared in [59]. We also show how this unification leads to a general instrument for pricing in column generation frameworks.

In chapter five, *The Job Shop Scheduling Problem*, we show how to incorporate bounding into the Dynamic Programming for the Job Shop Scheduling Problem and how to find all optimal solutions for JSSP instances. The computational results on the Job Shop Scheduling Problem in this chapter indicate that the complexity of this algorithm as proven in chapter 2 may possibly be improved. For example, by using some tighter bounds on the number of partial solutions created than those known to us. Also, it shows the effects of the bounded Dynamic Programming algorithm as well as the number of all optimal solutions for some small instances. We also show how Dynamic Programming with bounding can be used to validate or even find new lower bounds. For 16 out of the 97 unsolved Job Shop Scheduling Problem instances we were able to improve the lower bound.

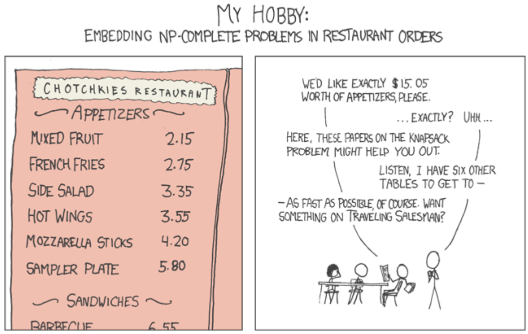
In the sixth and last chapter, *The Job Shop Scheduling Problem with Scheduled Maintenances*, we extend the Job Shop Scheduling Problem by adding maintenances on the machines. For this new problem we create a Mixed-Integer Programming formulation and we extend the Dynamic Programming algorithm to include these maintenances. We create a new bounding algorithm for this extension which can be used to improve the performance of the Dynamic Programming algorithm. We propose a method to generate instances for this extension, which appears to be very hard to tackle via Mixed-Integer Programming. However, with a bounded Dynamic Programming algorithm many of our generated instances could be solved to optimality.

Appendix A: *Computational Results* gives a detailed overview of all computational results as well as the information of the machine and software used. Appendix B: *Job Shop Scheduling Problem Instances* provides an overview of the Job Shop Scheduling Problem instances used in this thesis as well as their upper bounds and lower bounds. Also, the origin of these instances and their respective bounds are given in this appendix.

Finally, we want to point out the *Glossary of Notation* at the end of this dissertation. Part of this glossary can also be found on the inside of the cover flaps.




THE APOCALYPSE! THE SKIES BURN, THE SEAS TURN TO BLOOD, AND THE DEAD WALK THE EARTH!
THE DEAD WHAT?
WALK THE EARTH!
I HAVE TO GO.
SCRIBBLE
SCRIBBLE
THE DEAD RETURN!
EVERYONE, QUICK, GET YOUR NAMES ON HERE!
MATH DEPT
AT LAST!
I HOPE THERE'S TIME!
HURRGHHH
CEMETERY
PAUL ERDŐS?
YES?
WE NEED YOU TO SIGN THIS.



ONE

Dynamic Programming

This chapter introduces the basics of Dynamic Programming and presents the notation used throughout this dissertation, summarized in the *Glossary of Notation* (page 171). The second part of this chapter focusses on Dynamic Programming over sets. The famous Dynamic Programming algorithm for the Traveling Salesman Problem by Held and Karp [62] and Bellman [17] can be seen as an adaptation of this general principle for a particular problem.

1.1 The basics of Dynamic Programming

The basic idea behind Dynamic Programming (DP) is to split a problem recursively into simpler subproblems, the optimal solution to the problem can be easily found using the optimal solutions to these subproblems. The optimal solutions of these subproblems are found by splitting the subproblems again into even smaller problems, continuing until the solution of each subproblem is trivial. When an optimal solution of a (sub)problem can be found solely based on the optimal solutions of its subproblems the DP algorithm yields an optimal solution for the original problem. This is called the *Principle of Optimality* [see 16, chap. III.3.]. The recurrence relation that defines the relation between all the subproblems and how each problem is split up is called the Bellman equation. Ultimately, a DP algorithm amounts to solving the smallest trivial subproblems first, use their solutions to solve increasingly larger subproblems, until finally the complete problem is solved.

Intuitively, when we would calculate an optimal solution using a recurrence relation, we start with the whole problem, we then search recursively for the optimal solutions of the needed sub-problems, until the subproblems become trivial. This is the so-called backward algorithm. As stated above, we can also start with solutions for the trivial sub-problems and expand these to solutions for larger sub-problems, at each subproblem we take the best of the so created solutions, which is then optimal. Finally we arrive at the optimal solution of the whole problem. This is the so-called forward algorithm. For reasons soon to be explained, we focus our research on forward type of algorithms

The subproblems in the DP structure are called states. All states together form the so-called state space, this state space can be divided in stages containing states which represent sub-problems of the same size.

Definition 1.1
A sub-problem, or state, is defined by ξ_ϕ*. The subscript* ϕ *defines the specifics of the sub-problem.* □

Definition 1.2
Let ς *denote a solution. With* ς_ϕ *we define a solution to the sub-problem, or state,* ξ_ϕ*. By* $\check{\xi}_\phi$ *we denote an optimal solution to state* ξ_ϕ*.* □

Definition 1.3
By $\varsigma \diamond\!\!\Longrightarrow i$ *we denote an expansion in the forward DP algorithm from solution* ς *with* i*. This is a new solution of a larger sub-problem. The definition of* i *depends on the specific problem.* □

We can write the principle of optimality in terms of these state definitions.

Proposition 1.4
When the value of an optimal solution $\check{\xi}_\phi$ *for a state* ξ_ϕ *can be expressed using only the value of optimal solutions of other states* $\xi'_{\phi'}$ *and the expansion* i *such that* $\check{\xi}'_{\phi'} \diamond\!\!\Longrightarrow i$ *results in a solution in* ξ_ϕ*, the principle of optimality holds.* □

Proof This is essentially the principle of optimality where the definition of the state ξ_ϕ defines a sub-problem. ■

Corollary 1.5
If the principle of optimality holds, the feasibility of the expansion $\varsigma_\phi \diamond\!\!\Longrightarrow i$ *for a solution* $\varsigma_\phi \in \xi_\phi$ *only depends on* ξ_ϕ *and* i*, not on the specific solution* ς_ϕ*.* □

In the rest of this section we illustrate, using three simple problems, the basics of DP and some important properties.

1.1.1 Fibonacci numbers

One of the most well-known recurrence relations is the relation between the Fibonacci numbers. The Fibonacci numbers are defined by $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0, F_1 = 1$. These numbers are named after Leonardo Pisano, known as Fibonacci, who described them in 1202 [97], see [109] for a recent translation. The definition used here, starting with F_0, is from Lucas [80] who generalized this sequence. The recurrence relation follows directly from the definition, problem F_n can be found directly from the solutions of subproblems F_{n-1} and F_{n-2}. The relation between the different subproblems is described in figure 1.1, where the green and blue lines represent F_{n-1} and F_{n-2}, respectively.

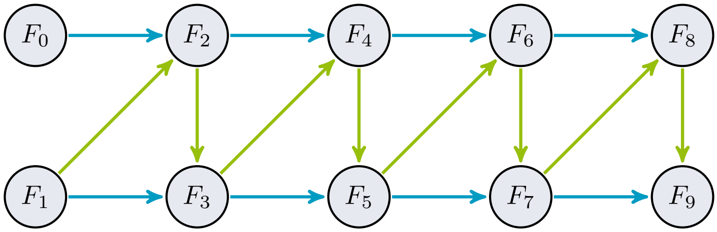


Figure 1.1: *Relation between the Fibonacci numbers*

Algorithm 1.1 DP algorithm for finding the n'th Fibonacci number

Input: A natural number n
Output: Fibonacci number F_n

$F_0 = 0$
$F_1 = 1$
for $i = 2$ **to** n **do**
    $F_i = F_{i-1} + F_{i-2}$

return F_n

1

These relations can always be described in a directed acyclic graph, whereas a cycle would include a subproblem whose solution depends on its own solution. This results in DP algorithm 1.1.

Note that a straightforward use of the recurrence relation as a recursive function, see algorithm 1.2, has complexity $\mathcal{O}(2^n)$ instead of $\mathcal{O}(n)$ of algorithm 1.1. This illustrates the difference between a recursive and a DP algorithm. The recursive algorithm will calculate the same value multiple times while the DP algorithm will save this value for later use. Naturally memoization, i.e., caching a previously calculated result to return later, will also result in an $\mathcal{O}(n)$ algorithm. Memoization in a recursive algorithm will result essentially in a backwards DP, while algorithm 1.1 is a forward DP algorithm. Later in this chapter we will take a closer look at the differences between forward and backward DP algorithms.

Algorithm 1.2 Recursive algorithm for finding the n'th Fibonacci number

Input: A natural number n
Output: Fibonacci number F_n

Fib(n)
    if $n = 0$ **or** $n = 1$ **then**
        return n
    else
        return **Fib**$(n-1)$ + **Fib**$(n-2)$

1.1.2 Longest common subsequence

The longest common subsequence problem is the problem of finding the longest subsequence occurring in two given sequences [121,18]. A subsequence is a sequence that can be constructed by deleting elements from the original sequence without changing the order. The subsequence does not have to be consecutive within the original sequence. One of the applications of the longest common subsequence problem is finding the longest common subsequence in two sequences of DNA. An example of two sequences with a common subsequence $\langle \mathrm{P}, \mathrm{T}, \mathrm{I}, \mathrm{A}, \mathrm{L} \rangle$ is

OPTIMAL
PICTORIAL

For the length of the longest common subsequence we can formulate a recurrence relation by looking at the last element in both sequences. Define $LCS(i,j)$ as the length of the longest common subsequence of the sequences consisting of the first i elements of the first sequence s_1 and the first j elements of the second sequence s_2. If the last elements are the same in both sequences, $s_1(i) = s_2(j)$, then we can remove this element from both sequences and find the longest common subsequence in the remaining sequences, thus $LCS(i,j) = LCS(i-1,j-1)+1$. If the last elements are different in both sequences, then that element can be removed from one of the sequences without changing the longest common subsequence, so $LCS(i,j) = LCS(i-1,j)$ or $LCS(i,j) = LCS(i,j-1)$, thus $LCS(i,j) = \max\{LCS(i-1,j), LCS(i,j-1)\}$. So the recurrence relation becomes

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1,j-1)+1 & \text{if } s_1(i) = s_2(j) \\ \max\{LCS(i-1,j), LCS(i,j-1)\} & \text{otherwise} \end{cases}$$

and the principle of optimality follows from the reasoning above. The complete values of the example above are given in figure 1.2.

s_1 \ s_2			P	I	C	T	O	R	I	A	L
	i \ j	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
O	1	0	0	0	0	0	1	1	1	1	1
P	2	0	**1**	1	1	1	1	1	1	1	1
T	3	0	1	1	1	**2**	2	2	2	2	2
I	4	0	1	2	2	2	2	2	**3**	3	3
M	5	0	1	2	2	2	2	2	3	3	3
A	6	0	1	2	2	2	2	2	3	**4**	4
L	7	0	1	2	2	2	2	2	3	4	**5**

Figure 1.2: *Longest Common Subsequence of OPTIMAL and PICTORIAL*

For the longest common subsequence a state is defined by i and j leading to a state definition of $\xi_{i,j}$, thus $\phi = (i,j)$. Note that the state space can be divided in equal sized sub-problems in two directions, i and j. For the longest common subsequence there are possibly three solutions associated with each state, for state $\xi_{i,j}$ these three solutions $\varsigma, \varsigma', \varsigma''$ are based on the optimal solutions $\check{\xi}_{i-1,j-1}, \check{\xi}_{i-1,j}, \check{\xi}_{i,j-1}$ of states $\xi_{i-1,j-1}, \xi_{i-1,j}, \xi_{i,j-1}$, respectively. The first solution ς originating from $\xi_{i-1,j-1}$ is only feasible if $s_1(i) = s_2(j)$. So the solution $\varsigma = \langle \check{\xi}_{i,j}, s_1(i) \rangle$ is the solution $\check{\xi}_{i,j}$ with the newly found common element $s_1(i)$ added, solutions ς' and ς'' are the same solutions as $\check{\xi}_{i-1,j}$ and $\check{\xi}_{i,j-1}$, respectively. Algorithm 1.3 describes a DP algorithm to find the longest common subsequence of two sequences, where the function *longest* selects the longest sequence. The complexity of this algorithm is $\mathcal{O}(mn)$, where m and n are the lengths of sequences s_1 and s_2, respectively. Note that the result of algorithm 1.3 is the longest common subsequence instead of its length given by the recursive function LCS described above.

Algorithm 1.3 DP algorithm for the longest common subsequence

Input: Two sequences s_1 and s_2
Output: The longest common subsequence of s_1 and s_2

m = length(s_1)
n = length(s_2)
for all $\check{\xi}_{i,j}$ **such that** $i = 0$ **or** $j = 0$ **do**
  $\check{\xi}_{i,j} = \langle\ \rangle$ // empty sequence

for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    if $s_i(i) = s_2(j)$ **then**
      $\check{\xi}_{i,j} = \langle \check{\xi}_{i-1,j-1}, s_1(i) \rangle$
    else
      $\check{\xi}_{i,j}$ = longest $\left(\check{\xi}_{i-1,j}, \check{\xi}_{i,j-1}\right)$

return $\check{\xi}_{m,n}$

Backtracking

In the complexity analysis of algorithm 1.3 the concatenation of the sequences is ignored. This would add an extra factor equal to the length of the resulting sequence, at most n or m. However, the length of the longest common subsequence can be found in $\mathcal{O}(mn)$ time by algorithm 1.3. The actual longest common subsequence can be found in $\mathcal{O}(m+n)$ by backtracking in the DP state space. We choose to represent a state by a sequence (solution), rather than by the length (cost) of a sequence, to be consistent in the representation of the states of all algorithms in this dissertation.

In order to find the longest common subsequence in the state space of a DP algorithm that finds the length of the longest common subsequence, we walk backwards through the created state space. We start at the resulting value in the last state, and move to the state which contributed to the value of the current state. The path through the state space of the example in figure 1.2 is marked with a green background. This path can be found by moving back in one of the original sequences to a state either above or on the left of the current state if the value is equal to the value of the current state. If the elements in both sequences are equal for the current state, we may move back in both sequences, thereby moving to the state above left of the current state, which value will be 1 lower as the value of the current state. All three moves may be available simultaneously, in which case multiple paths through the state space are feasible and all will result in an optimal solution. It is possible that multiple paths through the state space belong to the same optimal solution. With multiple optimal solutions the number of possible paths in the state space can explode. To obtain all optimal solutions efficiently DP can be used by using the state space of the first DP algorithm.

For a state space with a number of states that makes it practical to store in memory, such as for the longest common subsequence, backtracking is very applicable. However, for the exponentially large state spaces described in the following chapters this becomes impractical or even impossible. The complete state space must be saved during the algorithm to be able to traverse it later.

1.1.3 Knapsack

As figure 1.2 already shows a recurrence relation may define several states with the same solution. Finding the same solution over and over again for different states may however be avoided. To illustrate this we take a look at the Knapsack problem [83,71].

We have a knapsack that can hold a maximum total weight of W and we have n items, each item i has a weight w_i and a value v_i. The Knapsack problem consists of finding the items to carry as much value as possible without loading too much weight in our knapsack. There are two main variants of the Knapsack problem, with repetition — where we have unlimited copies for each item — and without repetition — with just one instance of each item — also called the 0–1 Knapsack problem. First assume that we have unlimited quantities of each item. Let $K(w)$ be the maximal value we can carry in a knapsack with maximum weight of w $(0 \leq w \leq W)$. We want to split this problem into subproblems. If item i is carried in the optimal solution then removing this item will result in the optimal solution for a smaller knapsack, thus $K(w-w_i) = K(w)-v_i$. If the value $K(w) - v_i$ would be non-optimal for $K(w - w_i)$, $K(w)$ would be non-optimal either; otherwise we could improve $K(w)$ by adding item i to the optimal solution resulting from $K(w - w_i)$. Since we do not know which item is in the optimal solution we have to try this for all items that fit in the current knapsack.

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\},$$

Algorithm 1.4 DP algorithm for the Knapsack

Input: A total weight W of the knapsack and a number of items n
For each item $i \in \{1, \ldots, n\}$ a weight w_i and a value v_i
Output: The items in the optimal Knapsack solution

$\check{\xi}_0 = \emptyset$

for $w = 1$ **to** W **do**
  $\check{\xi}_w = \emptyset$
  for $i = 1$ **to** n **do**
    if $w_i \leq w$ **and** $C(\check{\xi}_w) < C(\check{\xi}_{w-w_i}) + v_i$ **then**
      $\check{\xi}_w = \check{\xi}_{w-w_i} \cup \{i\}$

return $\check{\xi}_W$

is the resulting recurrence relation. The only state variable we need is w, i.e., a state is represented by ξ_w. The value of an optimal solution of a state is equal to the value of the recurrence relation $C(\check{\xi}_w) = K(w)$ where C is the function that returns the value, or cost, of a solution. Algorithm 1.4 describes the DP algorithm corresponding to this recurrence relation, which has a complexity of $\mathcal{O}(nW)$.

For the 0–1 Knapsack problem, when we have a single instance of each item, we cannot use the same relation, as we do not know if item i is already used in the optimal solution $\check{\xi}_{w-w_i}$. To keep track of which items are used, we not only look at smaller knapsacks, but also at fewer items. We define a state ξ_ϕ with $\phi = (w, j)$ which defines the 0–1 knapsack problem with maximum weight w using only items $1, \ldots, j$. The recurrence relation now becomes

Algorithm 1.5 Backward DP algorithm for the 0–1 Knapsack

Input: A total weight W of the knapsack and a number of items n
For each item $i \in \{1, \ldots, n\}$ a weight w_i and a value v_i
Output: The items in the optimal Knapsack solution

for all $\check{\xi}_{w,i}$ **such that** $w = 0$ **or** $i = 0$ **do**
  $\check{\xi}_{w,i} = \emptyset$

for $i = 1$ **to** n **do**
  for $w = 1$ **to** W **do**
    $\check{\xi}_{w,i} = \check{\xi}_{w,i-1}$
    if $w_i \leq w$ **and** $C(\check{\xi}_{w,i}) < C(\check{\xi}_{w-w_i,i-1}) + v_i$ **then**
      $\check{\xi}_{w,i} = \check{\xi}_{w-w_i,i-1} \cup \{i\}$

return $\check{\xi}_{W,n}$

1

$$C(\check{\xi}_{w,i}) = \max \left\{C(\check{\xi}_{w-w_i,i-1}) + v_i, C(\check{\xi}_{w,i-1})\right\},$$

where the first expression in the maximum is only used if $w_i \leq w$, otherwise the maximum weight of the knapsack would be exceeded. The cases represent the choice of putting item i in the knapsack and the choice of leaving item i out of the knapsack, respectively. The optimal solution, denoted by $\mathring{\xi}$, will be $\check{\xi}_{W,n}$. Algorithm 1.5 describes the DP algorithm according to this recurrence relation and has a complexity of $\mathcal{O}(nW)$. We see that the new state is initialized by leaving the item out of the knapsack, which is always feasible, this value is replaced by the choice of putting the item in the knapsack when this is feasible and better.

1

Item	Weight	Value
1	1	11
2	5	20
3	2	14
4	3	17

Let us now have a look at an example of a 0–1 Knapsack problem with 4 items and a maximum total weight of 6. When we look at optimal values of all states in figure 1.3, we see a lot of equal values. The same solution is often repeated.

w \ i	0	1	2	3	4
0	0 {}	0 {}	0 {}	0 {}	0 {}
1	0 {}	11 {1}	11 {1}	11 {1}	11 {1}
2	0 {}	11 {1}	11 {1}	14 {3}	14 {3}
3	0 {}	11 {1}	11 {1}	25 {1,3}	25 {1,3}
4	0 {}	11 {1}	11 {1}	25 {1,3}	28 {1,4}
5	0 {}	11 {1}	20 {2}	25 {1,3}	31 {3,4}
6	0 {}	11 {1}	31 {1,2}	31 {1,2,3}	42 {1,3,4}

Figure 1.3: *State space of backward DP for the 0–1 knapsack example*

We see for example that the optimal solution for all 4 items and a maximal weight of 4 ($\check{\xi}_{4,4}$) is 28, this is constructed from the values of $\check{\xi}_{4,3}$ and $\check{\xi}_{1,3}$, since the weight of item 4 is 3. This results in $28 = \max\{25, 11 + 17\}$.

1.1.4 Minimizing redundancy

The subproblems of the stage with only item 1, thus the states $\xi_{*,1}$, have only two possible solutions. Item 1 is either used or not, with values 11 and 0, respectively. However, we have to calculate the solutions for all 7 subproblems [see 6, chap. 7.3.].

To reduce the number of redundancies where the same solution is the optimal solution for different states, we redefine our states and do not solve the larger problems with optimal solutions of slightly smaller subproblems. Instead we use the optimal solutions of the smaller subproblems to find solutions for slightly larger problems. This looks the same, but it changes the focus from all the subproblems to the existing solutions. Essentially, this is the difference between calculating the DP state space in a backward or forward manner.

For the 0–1 knapsack problem we redefine the states $\xi_{*,i}$ to a single state ξ_i. However, we cannot have a single optimal solution per state anymore, we do have to keep multiple solutions per state. In state ξ_0 exists only a single solution with weight and value 0. As we know, the next state ξ_1 has two solutions based on the single solution in ξ_0. We find these solutions by expanding the solution in ξ_0 to two new solutions, by choosing whether to carry item 1 or not. State ξ_2 has now 4 solutions, by choosing wether to carry item 2 expanded from the two solutions in ξ_1. If we continue in this matter we have a brute force algorithm enumerating all possible solutions, so we have to find a way to safely ignore certain solutions. To achieve this we have to compare the solutions of a state and find solutions that are dominated by other solutions in the same state.

Definition 1.6
A completion of a partial solution is a solution of the original problem formed by a series of expansions until the last stage of the partial solution. □

Definition 1.7
One solution ς dominates an other solution ς' when the best completion of ς results in a better or equal solution than all completions of ς'. □

As example of domination we that we look at the solutions for the first three items. Start with the solution that takes only item 2, which is in the original state space state $\check{\xi}_{5,2}$. This can be expanded by choosing not to add item 3 into the knapsack, this solution still has a weight of 5 and a value of 20. However, we have also a solution taking items 1 and 3, in the original state space $\check{\xi}_{3,3}$, which has a weight of 3 and a value of 25. Both solutions are colored orange in the state space. So anything we can add into the first knapsack can in fact also be added into the second. There is even more space left and the value of the items carried is already higher. We conclude that we can discard the first solution, since it is dominated by the second. Note that choosing to add item 3 to the first solution $\check{\xi}_{5,2}$ is infeasible, since it will result in a weight of 7.

Until now, we had a single value, typically cost, to compare solutions within a single state. Now we have two relevant values to compare solutions within the same state, the value v and the weight w.

Definition 1.8
Let γ be an array of variables that are used to compare solutions in a state. We define the state as $\xi_{\phi,\gamma}$. The values of ϕ are the same for all solutions in this state. The values of γ may differ between these solutions. □

For the 0–1 Knapsack we use a new state definition with $\gamma = (v, w)$ and the fixed variable $\phi = (i)$, leading to a new state definition for $\xi_{\phi,\gamma}$. When ϕ and γ are not used as shorthand the variables represented by ϕ and γ are separated by $\wr$ resulting in $\xi_{i \wr v,w}$.

Until now, we had a single value to compare solutions within a state leading to a single optimal solution representing each state. Since we have now multiple values to compare solutions on, we cannot define a single optimal solution, instead we have possibly multiple non-dominated solutions. Within the same state all solutions are characterized by the same values for ϕ but possibly they have different values for γ, and we compare the variables of γ to find dominated solutions.

Definition 1.9
Define $\geqq$ as a pairwise comparison between the values of two arrays γ and γ'. We write $\gamma \geqq \gamma'$ when the values γ, for example (v, w), of one solution $\varsigma_{\phi,\gamma}$ dominate the values γ', for example (v', w'), of another solution $\varsigma'_{\phi,\gamma'}$. When two solutions ς, ς' do not dominate each other we write $\varsigma \not\doteq \varsigma'$ or therefore $\gamma \not\doteq \gamma'$. We write $\varsigma_{\phi,\gamma} \doteq \varsigma'_{\phi,\gamma'}$ when two solutions have equal values of γ, thus $\gamma \doteq \gamma'$ and therefore $\gamma = \gamma'$. □

In this case $\geqq$ is equal to $\{\geq, \leq\}$, i.e., the values of v and w are compared by $\geq$ and $\leq$, respectively. Thus, when ς dominates ς' we have $v \geq v'$ and $w \leq w'$. Note that the subscripts ϕ and γ are also used to describe properties of single solutions, $\geqq$ will also be used to describe solutions dominating each other, thus $\varsigma \geqq \varsigma'$ or $\varsigma_{\phi,\gamma} \geqq \varsigma'_{\phi,\gamma'}$.

Definition 1.10
We denote by $\hat{\xi}$ the set of non-dominated solutions within a state ξ in contrast to the optimal solution denoted by $\check{\xi}$. When there are multiple non-dominated solutions in ξ_i with $\varsigma \doteq \varsigma' \doteq \varsigma''$ only one of these solutions will be in $\hat{\xi}_i$, since we are interested in just one optimal solution. □

In figure 1.4 we see the total state space of the altered DP algorithm, as we can see each stage consists only of a single state $\xi_{\phi,\gamma}$, with $\phi = i$. Each state stores no longer a single optimal solution $\check{\xi}_{i,w}$ but a set of non-dominated solutions $\hat{\xi}_i$. The optimal solution will now be the solution $\varsigma_{n \wr v,w}$ with $v = \max_{\varsigma \in \hat{\xi}_n} C(\varsigma)$. Algorithm 1.6 describes the forward DP algorithm for the 0–1 Knapsack, note the differences with algorithm 1.5. The complexity of algorithm 1.6 is also $\mathcal{O}(nW)$. However, the typical running time would be lower, since not for all values of $w \leq W$ a non-dominated solution exists. One clear advantage occurs when a problem instance of the Knapsack is altered by multiplying all weights w_i and the maximum weight W with the same (integer) factor F, the running time of algorithm 1.5 is multiplied by F while the running time of algorithm 1.6 stays the same.

The fundamental difference between forward and backward calculation lies in the possibility for the forward calculation to evaluate solutions only when there is an actual choice to be made, while the backward calculation follows a predefined

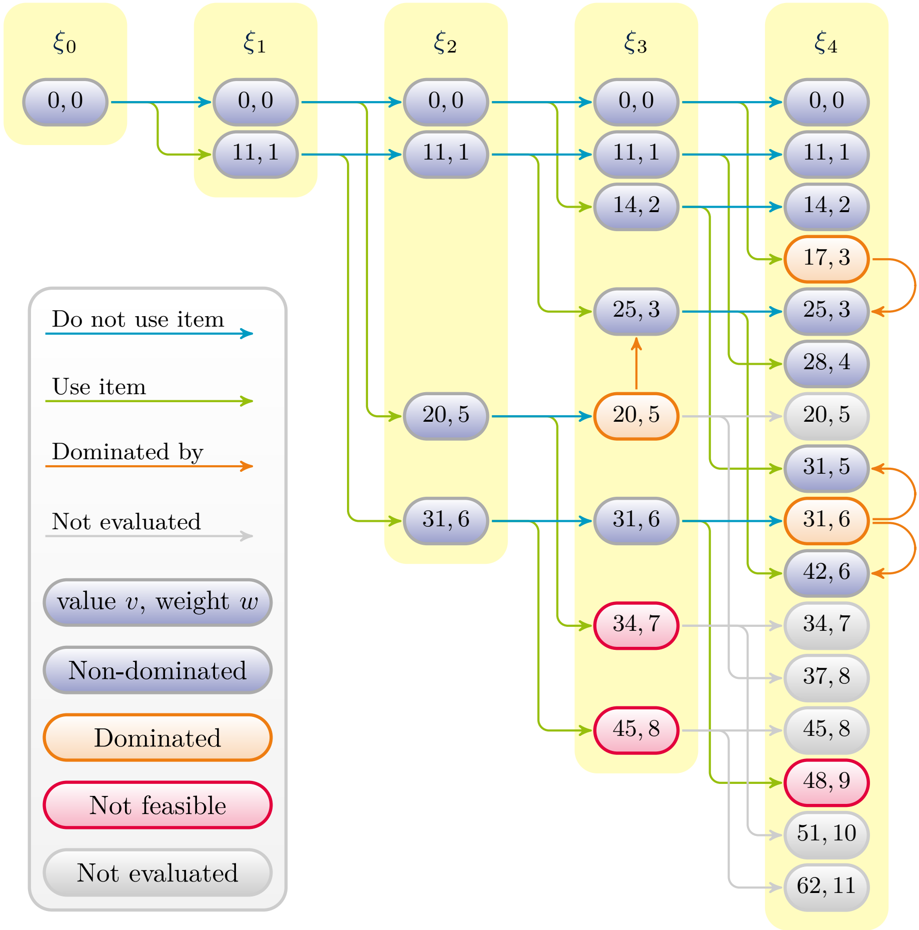


Figure 1.4: *State space of forward DP for the 0–1 Knapsack example*

path. The recurrence relation defines the cost for a state where all other variables are fixed. The recurrence relation as well as the backward calculation of the DP state space can have only one single variable, the cost, in the variables γ to compare within a single state. This leads to a single optimal solution for the state $\check{\xi}$. As the value of the cost, and thereby γ, is defined by the recurrence relation, these are left out of the state definitions in the recurrence relation. Since multiple variables in γ cannot effectively be used in the recurrence relation or backwards evaluation of the state space, the original recurrence relation stays the same. The forward calculation of the DP state space is just combining several states into a single state using several non-dominated solutions as representatives for a state instead of a single optimal solution. Using this method we can profit if a lot of solutions in different original states are actually the same solution or the range of a variable in the state definition is unknown leading to evaluating

Algorithm 1.6 Forward DP algorithm for the 0–1 Knapsack

Input: A total weight W of the knapsack and a number of items n
For each item $i \in \{1, \ldots, n\}$ a weight w_i and a value v_i
Output: The items in the optimal Knapsack solution

$\hat{\xi}_0 = \{\varsigma_0 \wr_{0,0} = \emptyset\}$

for $i = 1$ **to** n **do**
  for all $\varsigma_{i-1} \wr_{v,w} \in \hat{\xi}_{i-1}$ **do**
    // N represents the set of 1 or 2 new solutions
    $N = \{\varsigma_{i-1} \wr_{v,w}\}$ // Leave item i out of the knapsack
    if $w + w_i \leq W$ **then**
      $N = N \cup \{\varsigma_{i-1} \wr_{v,w} \cup \{i\}\}$ // Adding item i to the knapsack
    for all $\varsigma \in N$ **do**
      if $\varsigma \not\lesseqgtr \varsigma_i, \forall \varsigma_i \in \hat{\xi}_i$ **then**
        $D = \left\{ \varsigma_i \in \hat{\xi}_i \;\middle|\; \varsigma \gtrsim \varsigma_i \right\}$ // All solutions in $\hat{\xi}_i$ dominated by ς
        $\hat{\xi}_i = \{\varsigma\} \cup \hat{\xi}_i \setminus D$

$\mathring{\varsigma} \in \left\{ \varsigma \in \hat{\xi}_n \;\middle|\; C(\varsigma) = \max_{\varsigma \in \hat{\xi}_n} C(\varsigma) \right\}$

return $\mathring{\varsigma}$

all states for all possible values of that particular variable. Note that for a state definition $\xi_{\phi,\gamma}$ the set of dominated solutions for a state $\xi_{\phi,\gamma}$ is $\hat{\xi}_\phi$, as all values of ϕ are equal for the corresponding solutions while the values of γ may vary, this corresponds to the removal of the cost in the recurrence relation, which is the only variable in γ for the backward calculation.

Proposition 1.11
If the set of non-dominated solutions $\hat{\xi}_\phi$ for a state $\xi_{\phi,\gamma}$ can be expressed using only:

- *the value of non-dominated solutions of other states $\xi'_{\phi',\gamma'}$,*
- *the expansion i such that $\hat{\xi}'_{\phi'} \Leftrightarrow\!\!\Rightarrow i$ results in a solution in $\xi_{\phi,\gamma}$.*

If also $\gtrsim$, and thereby the choice of γ, is defined such that a dominating value always lead to equal or more slack than the dominated values, when making the same expansions. Then the principle of optimality holds. □

Proof Follows from proposition 1.4 and the state domination defined by $\gtrsim$. This prevents that a solution can be dominated for which there exists a feasible expansion such that the same expansion is infeasible for the dominating solution. In other words, no set of expansions leads to an infeasible solution when expanded from the dominating solution, while the same set of expansions leads to a feasible solution from the dominated solution. ■

Similar to corollary 1.5 the feasibility only depends on the values of ϕ and γ and the expansion, not on the expanded solution.

1.2 Dynamic Programming over sets

Since all DP algorithms in this dissertation are done over sets, we show three examples of DP algorithms over sets, before starting with the famous DP algorithm for the Traveling Salesman Problem of Held and Karp [62] and Bellman [17] in the next chapter. We show examples of DP algorithms for the following three problems

- Linear assignment problem
- Steiner tree in graphs
- Single machine total weighted tardiness scheduling problem

The algorithms we present here are chosen for their example value, they are not chosen for efficiency. The Linear assignment problem can be solved in polynomial time [74], the last two are NP-hard [50]. However, all three DP algorithms use exponential time to solve these problems. A similar DP algorithm for the Linear assignment problem is also used as example in [68, chap. 2]. The DP algorithm for the Steiner tree in graphs in this section should not be confused with known and more efficient DP algorithms for the Steiner tree in graphs as given in [39,47,48]. The Single machine total weighted tardiness scheduling problem described in this section contains also release times ($1|r_j|\sum w_j T_j$ [see 58]), in [1] a similar DP algorithm can be found without release times.

The Linear assignment problem shows DP over sets, while Steiner tree in graphs and the Single machine total weighted tardiness scheduling problem show the difference between forward and backward calculation of the DP algorithm. With the Steiner tree in graphs we do not know the state that will result in the optimal solution while with the Single machine total weighted tardiness scheduling problem we do not know beforehand the possible values for a state variable.

1.2.1 Linear assignment problem

The linear assignment problem aims at finding a bijection between two sets of equal size, with minimal cost. For example we have a project with a set T of n tasks and a set E of n employees, i.e., $|T| = |E| = n$. Not every employee can handle every task with the same efficiency, so for each combination of task and employee we have a cost $c(t,e)$ ($t \in T$, $e \in E$). Now we have to find the assignment $a : T \to E$ such that the total cost of the project $C = \sum_{t \in T} c\left(t, a(t)\right)$ is minimized. More information on assignment problems can be found in [23].

To create the DP algorithm we first define an order $t_1, \ldots, t_n$ in which the tasks will be assigned. Now we create a state definition of $\xi_{S \wr c}$ where $S \subseteq E$ is the set of employees already assigned a task and c is the sum of the cost of the current partial assignment. An expansion of the optimal solution $\check{\xi}_S$ of state ξ_S will be the assignment of an employee $e \in E \setminus S$ to task $t_{|S|+1}$. Note that we have a single optimal solution $\check{\xi}$ per state ξ as γ consists of a single element, the cost. The recurrence relation now becomes

$$C\left(\check{\xi}_S\right) = \min_{i \in S} \left\{ C\left(\check{\xi}_{S \setminus \{i\}}\right) + c\left(i, t_{|S|}\right) \right\}.$$

As we can see, the new cost only depends on the cost of previous states and the added cost only on the employees i in S and its current size $|S|$. Note that only the set S and its properties are used, not the sequence that represents a solution. While using forward calculation, instead of the backwards formulation of the recurrence relation, a solution $\varsigma_{S \wr c} = \check{\xi}_S$ is expanded $(\varsigma_{S \wr c} \Diamond\!\!\Rightarrow i)$ into a new solution $\varsigma_{S \cup \{i\} \wr c + c(i, t_{|S|+1})}$ for every $i \in E \setminus S$. These solutions will be dominated in their corresponding states $\xi_{S \cup \{i\}}$ on the value of c. Since $|\gamma| = 1$ we have a single optimal solution for each state and the same states are evaluated as with the backwards formulation. Naturally we start whith the empty solution $\varsigma_{\emptyset \wr 0} = \check{\xi}_\emptyset$, and find the optimal solution $\mathring{\varsigma} = \check{\xi}_E$.

Algorithm 1.7 A DP algorithm for the linear assignment problem

Input: Sets of tasks $T = \{t_1, \ldots, t_n\}$ and employees $E = \{e_1, \ldots, e_n\}$
A cost $c(t_i, e_j)\ \forall t_i \in T, e_j \in E$

Output: A sequence of employees, where e_j at position i indicates that task t_i is handled by employee e_j

Let $\varsigma_{\emptyset \wr c}$ be the solution with an empty sequence $\langle\ \rangle$ and $c = C(\varsigma) = 0$
$\check{\xi}_\emptyset = \varsigma_{\emptyset \wr 0}$

for $L = 0$ **to** $n - 1$ **do**
  for all $S \subset E$ **such that** $|S| = L$ **do**
    for all $e_j \in E \setminus S$ **do**
      if $\check{\xi}_{S \cup \{e_j\}} = \emptyset$ **or** $C(\check{\xi}_S) + c(t_{L+1}, e_j) < C(\check{\xi}_{S \cup \{e_j\}})$ **then**
        $\check{\xi}_{S \cup \{e_j\}} = \check{\xi}_S \Diamond\!\!\Rightarrow e_j$ // $= \langle \check{\xi}_S, e_j \rangle$

return $\check{\xi}_E$

Algorithm 1.7 describes a DP algorithm for the linear assignment problem. The optimality principle holds as the choice expansions, the new values of all

	t_1	t_2	t_3	t_4
e_1	16	3	2	13
e_2	9	6	7	12
e_3	5	10	11	8
e_4	4	15	14	1

Table 1.1: *Small instance of the linear assignment problem*

state variables of all expansions depend solely on the previous state variables S and c, and the choice of the expansion i. Note that we also use $|S|$ which is just a property of state variable S. The complexity of the algorithm is $\mathcal{O}(n2^n)$, each state is expanded to at most n new states in $\mathcal{O}(1)$ time and there are 2^n

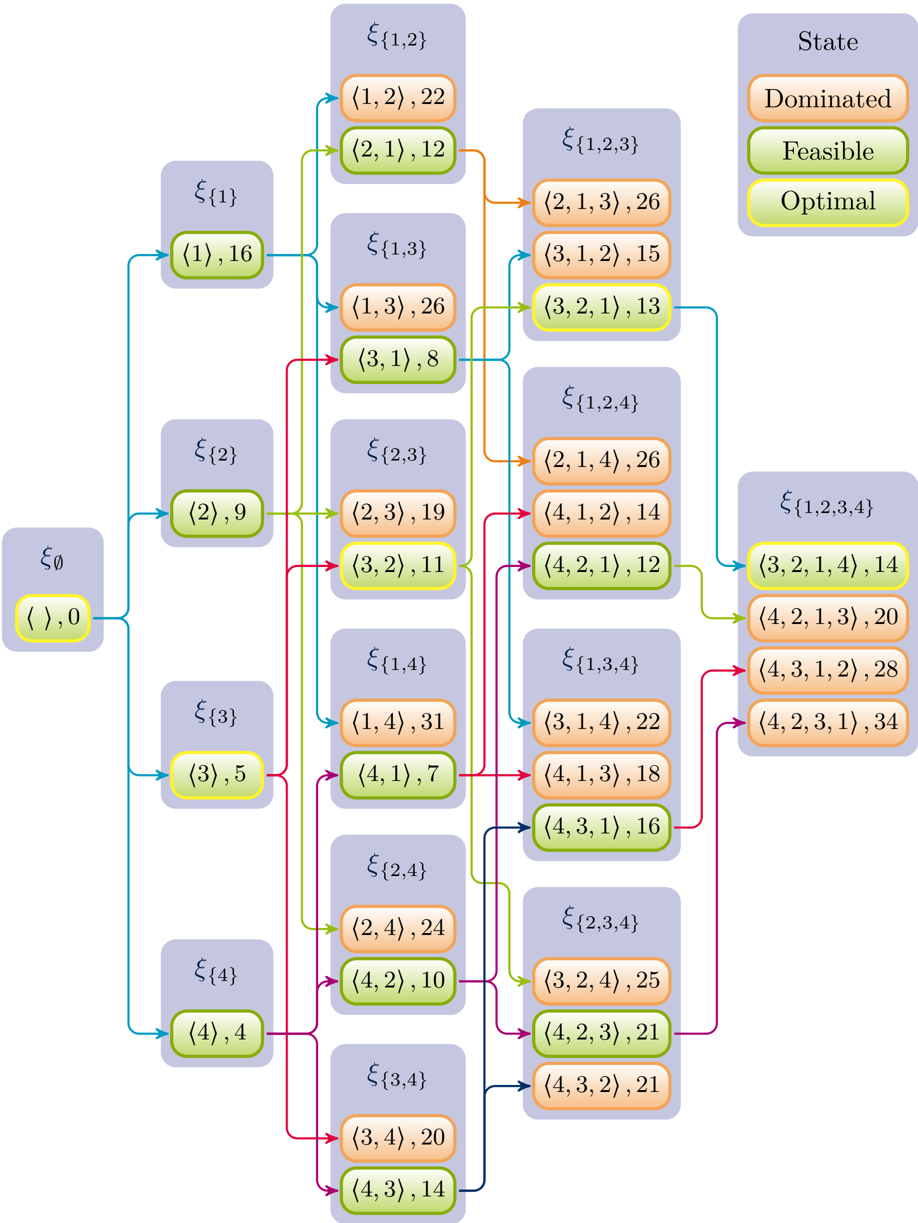


Figure 1.5: *State space of DP for the linear assignment problem in table 1.1*

possible states, since there are 2^n subsets of E. In fact a constant factor 2 may be removed from this complexity by noticing that there are only $|E| - |S|$ possible node to expand to and $\sum_{k=0}^{n}(n-k)\binom{n}{k} = n2^{n-1}$.

To illustrate the DP algorithm over sets we take a look at the DP state space of a small example of the linear assignment problem, with costs as given in table 1.1. The DP state space for this small example is depicted in figure 1.5.

1.2.2 Steiner tree in graphs

In our next example we show another advantage of the forward DP algorithm. Sometimes it is not known in which state holds the optimal solution, this in contrast to for example the linear assignment problem of the previous section where the state ξ_S provides the optimal solution. Furthermore, forward calculation may be an advantage when we have infeasible solutions.

The Steiner tree problem in graphs consists of an undirected graph $G = (V, E)$ weight $w(e) \geq 0$ for all edges $e \in E$ and a subset $R \subseteq V$ of required vertices [39]. The goal is to find the connected subgraph of G which includes all vertices of R such that the total weight of all edges is minimal. Note that this graph can by definition be reduced to a tree.

For the Steiner tree problem we create a state definition of $\xi_{S \wr w}$ where $S \subseteq V$ and w is the total weight of a tree spanning S. Every solution $\varsigma_{S \wr w}$ will be a, not necessarily minimal, spanning tree of S. However, the optimal solution $\check{\xi}_S$ will be the minimal spanning tree of S. A solution $\varsigma_{S \wr w} = \check{\xi}_S$ ($|\gamma| = 1$) is expanded with i ($\varsigma_{S \wr w} \diamond\!\!\Rightarrow i$) into a new solution $\varsigma_{S \cup \{i\} \wr w + f(S,i)}$ for every $i \in V \setminus S$. Here, $f(S, i)$ is defined as $\min_{j \in S | e=(i,j) \in E} w(e)$, which is the minimal weight of any edge connecting i with S. The expanded solution $\varsigma_{S \cup \{i\} \wr w + f(S,i)}$ represents the tree represented by $\varsigma_{S \wr w}$ with the addition of this minimal connecting edge and the expanded solution becomes infeasible when no such edge exists, and it is discarded. For the backward calculation discarding infeasible solutions is not possible, since the cost of every state used in the recurrence relation must be known. To cope with this, infeasible solutions can be assigned a cost of ∞. However, all states must be evaluated, even if there are no feasible solutions for a state. The graph represented by the solutions will always represent a tree, since the edge added by an expansion is always an edge to a new vertex. This prevents the creation of cycles. The corresponding recurrence relation is

$$C\left(\check{\xi}_S\right) = \min_{i \in S}\left\{C\left(\check{\xi}_{S \setminus \{i\}}\right) + f\left(S \setminus \{i\}, i\right)\right\}.$$

Again we start with the empty solution $\varsigma_{\emptyset \wr 0} = \check{\xi}_{\emptyset}$. However, we do not know beforehand which state holds the optimal solution. This is not necessarily in $\check{\xi}_V$, since this is the minimal spanning tree of G. The optimal solution is also not in necessarily $\check{\xi}_R$. For example, when R is not necessarily connected in G, in this case $\check{\xi}_R$ does not exist. The weight of the optimal solution is in this case

$$\min_{R \subseteq S \subseteq V} C\left(\check{\xi}_S\right).$$

During the backward calculation we have to explore all states, i.e., finding the minimum over all possible subsets $S \setminus \{i\}$. However, during the forward calculation the expansion over non-existing edges is just skipped, we can also stop expanding any solution $\check{\xi}_S$ when $R \subseteq S$.

Again all new values of the state variables of an expansion can be calculated using the state values of the expanded solution and the vertex which is used to expand the solution. The underlying spanning tree represented by a solution is not used during the expansion or the test for the feasibility, the set S is sufficient to find the minimal edge. In fact, the optimal solution $\check{\xi}_S$ of state ξ_S represents the minimal spanning tree of S. Since the edges added in the order of Prim's Algorithm [98] will form a minimal spanning tree in each stage, the solution representing a minimal spanning tree cannot be dominated in earlier stages.

1

Algorithm 1.8 A DP algorithm for the Steiner tree problem in graphs

---

Input: A graph $G(V, E)$ with a weight $w(e) \forall e \in E$
A set $R \subset V$

Output: A sequence which describes a subset $S \supseteq R$ of V such that the minimal spanning tree of S is the minimal subtree of G containing all vertices R

$\check{\xi}_\emptyset = \varsigma_{\emptyset \wr 0} = \langle \ \rangle$ **and** $\mathring{\varsigma} = \emptyset$

for $L = 0$ **to** $|V| - 1$ **do**
  for all $S \subset V$ **such that** $|S| = L$ **and** $R \nsubseteq S$ **do**
    for all $i \in V \setminus S$ **do**
      if $S \cup \{i\}$ is connected in G **then** // Feasibility
        if $R \subseteq S \cup \{i\}$ **then** // Test the best solution
          if $\mathring{\varsigma} = \emptyset$ **or** $C(\check{\xi}_S) + f(S, i) < C(\mathring{\varsigma})$ **then**
            $\mathring{\varsigma} = \check{\xi}_S \diamond\!\!\Rightarrow i$ // $= \langle \check{\xi}_S, i \rangle$
        else
          if $\check{\xi}_{S \cup \{i\}} = \emptyset$ **or** $C(\check{\xi}_S) + f(S, i) < C(\check{\xi}_{S \cup \{i\}})$ **then**
            $\check{\xi}_{S \cup \{i\}} = \check{\xi}_S \diamond\!\!\Rightarrow i$ // $= \langle \check{\xi}_S, i \rangle$

return $\mathring{\varsigma}$

Algorithm 1.8 describes the forward DP algorithm. Again the complexity is $\mathcal{O}(n 2^n)$ for n possible expansions for each of the 2^n subsets of V. Not all sets are expanded, since sets $S \supseteq R$ are not expanded. This gives a reduction of $2^{|V| - |R|}$ of the $2^{|V|}$ sets, which does not affect the theoretical worst-case time complexity. The subset S of V $(R \subseteq S \subseteq V)$ described by the returned sequence $\mathring{\varsigma}$ is sufficient to efficiently reconstruct the minimal spanning tree of S in G. The sequence $\mathring{\varsigma}$ gives a possible order of the vertices as they could be found by Prim's Algorithm to construct the minimal spanning tree containing all vertices in $\mathring{\varsigma}$.

In this DP algorithm there may be many states that do not have any feasible solution.

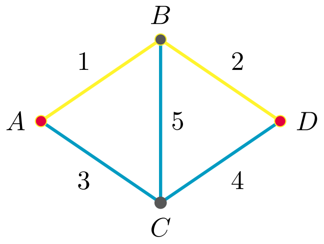


Figure 1.6: *A small Steiner tree instance*

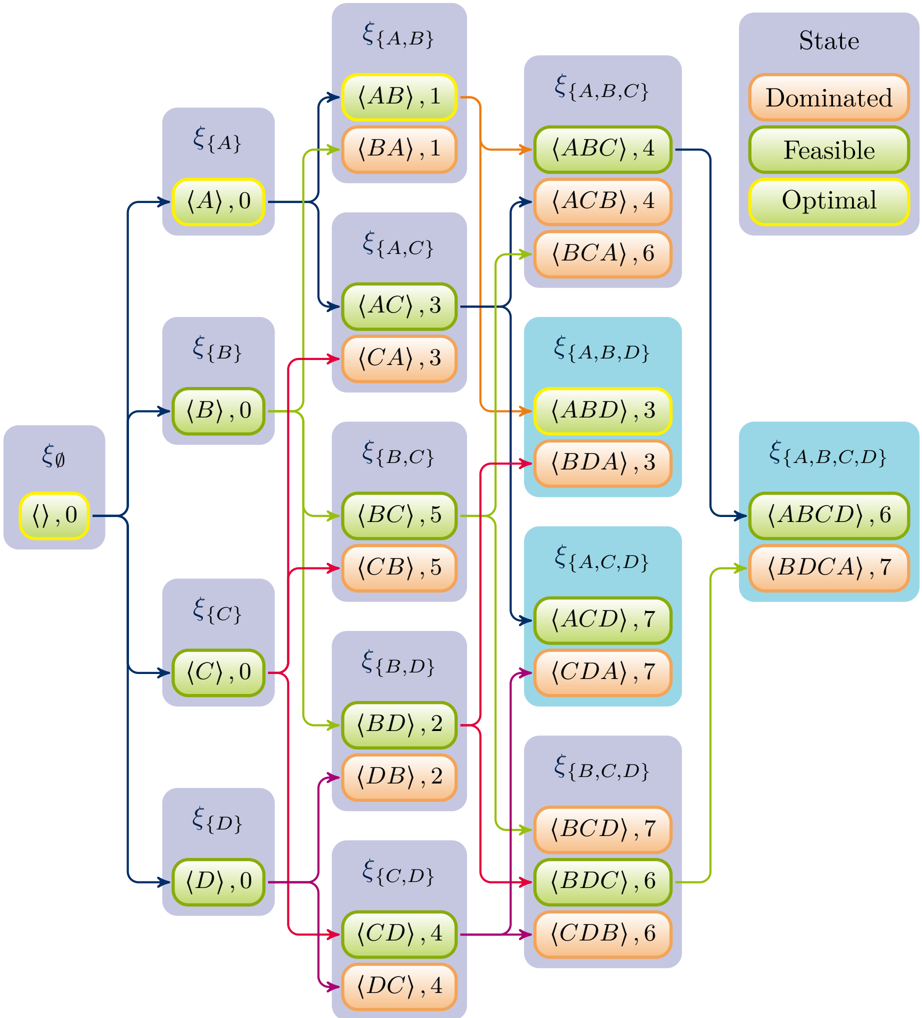


Figure 1.7: *State space of DP for the Steiner tree in graph problem of figure 1.6*

In the forward calculation we do not notice this as there are just no solutions feasibly expanded into those states and so they are not evaluated. During the backward calculation this is noticed as these states are evaluated and their solutions have a cost of ∞. Feasible solutions do not exist for a state ξ_S when the vertices S are not connected in G. Since this is dependant on state variable S, in this case it is possible to incorporate it into the recurrence relation and the backward calculation. However, this is in general not necessarily possible.

We illustrate this DP algorithm with a small example graph of four nodes, with nodes $V = \{A, B, C, D\}$ and required vertices $R = \{A, D\}$. In the graph of figure 1.6 the required nodes are red and the optimal solution is marked with yellow. The DP state space for this small instance is given in figure 1.7. As soon as the required vertices are considered no expansions are needed any more, since its spanning tree is a Steiner tree and adding extra edges will increase the cost. The states including the required vertices are marked blue in figure 1.7, the best solution over all these states is the optimal solution. In this small example the DP algorithm can be stopped after stage 4, solution $\langle ABD \rangle$ has value 3 and the only solutions that can be expanded in stage 4 $\langle ABC \rangle$ and $\langle BDC \rangle$ have already higher values. Moreover, it would be sufficient to start with just one of the required vertices instead of all vertices, since this vertex will be in the Steiner tree and for Prim's Algorithm it is sufficient to start with any vertex.

1

1.2.3 Single machine total weighted tardiness problem

In our final example we take a look at a DP algorithm over sets which uses multiple variables to dominate, thus $|\gamma| > 1$. For this we take a look at scheduling tasks on a single machine. We have a set of tasks T, and for each task $t \in T$ we have a length $l(t)$, a weight $w(t)$, a release time $r(t)$ and a deadline $d(t)$, with $l(t), w(t), r(t), d(t) \in \mathbb{N}_0$. Now we have to schedule these tasks for a single machine in such a way that these tasks do not overlap and the total weighted tardiness is minimized. That is, find a start time $\sigma(t) \geq r(t)$ for each task such that the intervals $(\sigma(t), \sigma(t) + l(t))$ do not overlap for any two tasks and $\sum_{t \in T: \sigma(t)+l(t)>d(t)} (\sigma(t) + l(t) - d(t)) \cdot w(t)$ is minimized.

For our DP algorithm we create a state definition of $\xi_{S \wr w,\tau}$, where $S \subseteq T$ are the tasks already scheduled, τ is the latest end time of all tasks in S, and w is the total weighted tardiness. To create a recurrence relation we can only have a single variable in γ. Since we want to minimize w, we have to move τ to the fixed state variables ϕ. This results in a state definition of $\check{\xi}_{S,\tau}$ and with $C\left(\check{\xi}_{S,\tau}\right) = w$ we get the recurrence relation

$$C\left(\check{\xi}_{S,\tau}\right) = \min \Bigg\{ C\left(\check{\xi}_{S,\tau-1}\right), \\ \min_{\{\, i \in S \;|\; \tau - l(i) \geq r(i) \,\}} \left\{ C\left(\check{\xi}_{S \setminus \{i\}, \tau - l(i)}\right) + w(t) \cdot \frac{\tau - d(i) + |\tau - d(i)|}{2} \right\} \Bigg\},$$

where the rightmost part evaluates to 0 if $\tau < d(i)$ and to $w(i) \cdot (\tau - d(i))$ otherwise.

However, we do not know the end time of the optimal schedule so we have to calculate $C\left(\check{\xi}_{T,\tau}\right)$ for every τ to find the optimal schedule. Logically some good estimations can be made to limit the values of τ that need to be calculated; however, this is not the only problem. For every intermediate state $\check{\xi}_{S,\tau}$ it is unclear whether there is any feasible solution. It is possible that we have to evaluate a lot of extra states to reach a negative conclusion. Without the release times there are no idle times in the optimal schedule and for $\check{\xi}_{S,\tau}$ we would get $\tau = \sum_{i \in S} l(i)$, this leads to the DP algorithm of Schrage and Baker [107]

The forward calculation finds the possible values for τ automatically. A solution $\varsigma_{S \wr w,\tau} \in \hat{\xi}_S$ is expanded with i ($\varsigma_{S \wr w,\tau} \Diamond\!\!\Rightarrow i$) to $\varsigma_{S \cup \{i\} \wr w',\tau'}$, with $\tau' = \max\{\tau, r(i)\} + l(i)$ and $w' = w + w(i) \cdot \frac{\tau' - d(i) + |\tau' - d(i)|}{2}$. Furthermore, since we want to minimize w and a lower value of τ gives more slack, we have $\geqq = \{\leq, \leq\}$, thus γ dominates γ' ($\gamma \geqq \gamma'$) if $w \leq w'$ and $\tau \leq \tau'$. The empty solution we start with is $\varsigma_{\emptyset \wr 0,0}$ as $\left\{\varsigma_{\emptyset \wr 0,0}\right\} = \hat{\xi}_\emptyset$. The optimal weighted tardiness can now be found among the solutions $\hat{\xi}_T$ by $\min_{\varsigma \in \hat{\xi}_T} C(\varsigma)$. This is described in algorithm 1.9.

Note that the optimal solution has to be found in $\hat{\xi}_T$ looking for the lowest value of w, disregarding the values of τ. The algorithm has a complexity of $\mathcal{O}\left(U^2 n 2^n\right)$, where U is an upper bound on the number of non-dominated solutions in any state. One factor of U is due to extra expansions from a single state, while another factor of U is due to a possible comparison of a new solution against

Algorithm 1.9 A DP algorithm for Single machine total weighted tardiness scheduling problem

Input: A set tasks T
For all $t \in T$ a length $l(t)$, a weight $w(t)$, a release time $r(t)$ and a deadline $d(t)$ (to be used during the expansion)
Output: The optimal sequence in which the tasks should be scheduled

$\hat{\xi}_\emptyset = \{\varsigma_{\emptyset \wr 0,0} = \langle\ \rangle\}$

for $L = 0$ **to** $|T| - 1$ **do**
  for all $S \subset T$ **such that** $|S| = L$ **do**
    for all $\varsigma_{S \wr w,\tau} \in \hat{\xi}_S$ **do**
      for all $i \in T \setminus S$ **do**
        $\varsigma = \varsigma_{S \wr w,\tau} \Diamond\!\!\Rightarrow i$
        if $\varsigma \not\leqq \varsigma', \forall \varsigma_i \in \hat{\xi}_{S \cup \{i\}}$ **then**
          $D = \left\{ \varsigma' \in \hat{\xi}_{S \cup \{i\}} \,\middle|\, \varsigma \geqq \varsigma' \right\}$ // All solutions dominated by ς
          $\hat{\xi}_{S \cup \{i\}} = \{\varsigma\} \cup \hat{\xi}_{S \cup \{i\}} \setminus D$

$\mathring{\varsigma} \in \left\{ \varsigma \in \hat{\xi}_T \,\middle|\, C(\varsigma) = \min_{\varsigma \in \hat{\xi}_T} C(\varsigma) \right\}$

return $\mathring{\varsigma}$

all other solutions in a state. When a new solution is tested for domination by existing solutions in a state a factor U can be reduced to $\log U$. However, when the new solution could be added still all existing solutions in the state should still be checked on domination by the new solution. As an upper bound we can take $U = \sum_{t \in T} l(t) + \max_{t \in T} r(t)$, since this is the total time to schedule all tasks after the last release time, although typically just a few non-dominated solutions exist per state.

	$l(t)$	$w(t)$	$r(t)$	$d(t)$
t_1	5	4	1	8
t_2	3	1	0	9
t_3	5	10	2	7

Table 1.2: *Small instance of a Single machine total weighted tardiness scheduling problem*

We illustrate the difference between the forward and the backward calculation with a small example given in table 1.2. Figure 1.8 gives the state space of the forward DP algorithm while figure 1.9 gives the state space of the backward DP algorithm. Even for this small example we see that the forward DP uses 15 states while the backward DP uses at least 20 states, which is only reached when some feasibility check is added to eliminate the consideration of the 56

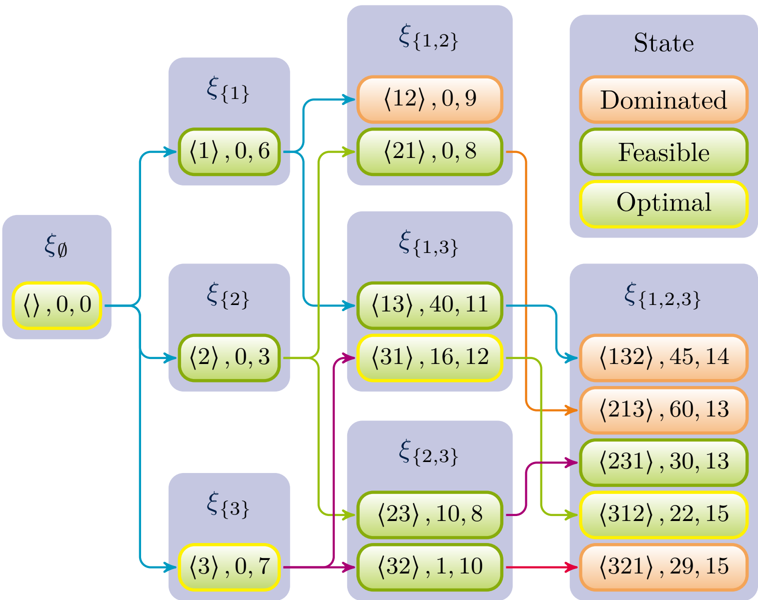
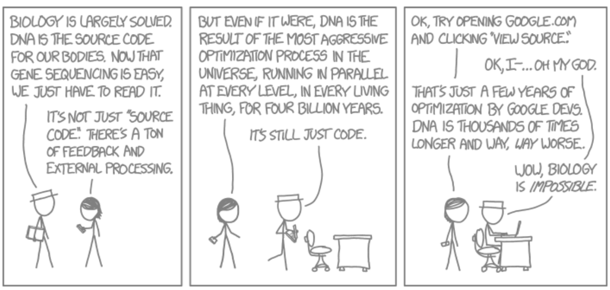


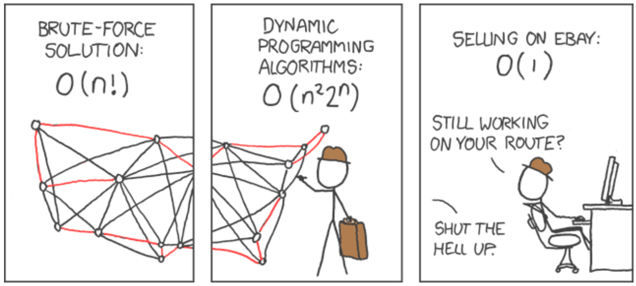
Figure 1.8: *State space of the forward DP for the Single machine total weighted tardiness scheduling problem of table 1.2*

infeasible states.

τ \ S	$\emptyset$	$\{1\}$	$\{2\}$	$\{3\}$	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$
0	0	-	-	-	-	-	-	-
1	0	-	-	-	-	-	-	-
2	0	-	-	-	-	-	-	-
3		-	0	-	-	-	-	-
4		-	0	-	-	-	-	-
5		-	0	-	-	-	-	-
6		0		-	-	-	-	-
7		0		0	-	-	-	-
8					0	-	10	-
9					0	-	10	-
10					1	-	1	-
11						40		-
12						16		-
13								30
14								30
15								22

Figure 1.9: *State space of the backward DP for the Single machine total weighted tardiness scheduling problem of table 1.2*





TWO

Sequencing, Routing and Scheduling

In this chapter we look at the classic variant of each of the three NP-hard problems addressed in this dissertation.

- Traveling Salesman Problem
- Vehicle Routing Problem
- Job Shop Scheduling Problem

The Traveling Salesman Problem was one of the first problems solved by DP over sets in 1962. We extend this algorithm to the Vehicle Routing Problem, which is the extension of the Traveling Salesman Problem using multiple routes, see also [59]. To solve the Job Shop Scheduling Problem with DP we carefully construct multiple state variables to be able to schedule jobs over multiple machines simultaneously, see also [60].

In the previous chapter we have seen the basic concepts of DP over sets. The common denominator of DP over a set $\mathcal{S}$ is that for each subset $S \subseteq \mathcal{S}$ we want to find the optimal solution $\varsigma_S = \check{\xi}_S$ for state ξ_S. Each solution ς_S is represented by a sequence of all nodes in S, finally leading to an optimal sequence $\mathring{\varsigma}_{\mathcal{S}}$ over all nodes $\mathcal{S}$. Furthermore, the expansion from one solution to another is done by adding a single node at the end of the sequence.

When we have the most basic state definition of $\xi_{S\wr c}$, with some cost c and no other state variables, we have a total of 2^n, with $n = |\mathcal{S}|$, possible states, since we have 2^n possible subsets $S \subseteq \mathcal{S}$. Each state ξ_S can be expanded to $|\mathcal{S}| - |S|$ nodes which is at most n possible nodes. Assuming that each expansion, including feasibility check and comparison with the current best of the state that is expanded to, the DP algorithm over sets has a computational complexity of $\mathcal{O}(n2^n)$. This is exponentially better than evaluating all possible orders of nodes in $\mathcal{S}$, since there are $n!$ possible sequences of all nodes in $\mathcal{S}$ the complexity thereof will be $\mathcal{O}(n!)$. Note that the memory requirements of a DP algorithm over sets is also exponential, the middle stage consisting of states ξ_S with $|S| = \frac{1}{2}n$ has size 2^{n-1}, since there are 2^{n-1} subsets $S \subset \mathcal{S}$ with $|S| = \frac{1}{2}n$. At least two stages need to be kept in memory, so the memory requirement is $\mathcal{O}(2^n)$.

2.1 Traveling Salesman Problem

A description of the Traveling Salesman Problem (TSP) is easily given: visit a collection of cities exactly once using the shortest possible route, returning to the city the route started in. More formally, given a complete graph $G = (V, E)$ with a distance c_{ij} for all edges $e_{ij} \in E$, find the shortest cyclic path visiting all vertices $v \in V$. Although the TSP can be described with ease, solving an instance of the TSP to optimality can be very hard. Currently the largest instance ever solved has 85 900 vertices and is first solved in 2006 [4,5]. The solution proved optimal in 2006 was already found by Helsgaun using Lin-Kernighan heuristic in 2004, see also [8,63,64,65]. It took approximately 136 CPU-years to prove the optimality of this solution. More information on the TSP can be found in [78,104,61,4,34].

A solution for the TSP can start at any node s as the tour of a TSP solution is cyclic. So we select a start vertex $s \in V$ to start the TSP solution. Assume we would use a state definition of $\xi_{S \wr c}$. We would have a solution that would be the shortest path starting in s and visiting all vertices $S \subseteq V$. However, when expanding a solution $\varsigma_S = \check{\xi}_S$ to a vertex $i \in V \setminus S$ we should know the distance c_{li} from the last vertex l in the path to i. This depends on the last vertex of the solution ς_S which is not in the state definition $\xi_{S \wr c}$. To be able to use this last vertex we should add it to the state definition, which becomes $\xi_{S,l \wr c}$. A solution for state $\xi_{S,l}$ now represents a path from s to l visiting all vertices in S. Note that we say a path visited a vertex i if the path traveled to i, so $l \in S$ and typically $s \notin S$. This means that we start our DP algorithm with state $\check{\xi}_{\emptyset,s} = \varsigma_{\emptyset,s \wr 0}$. Since the path starts in s, the path needs also to finish in s thus the optimal solution $\mathring{\varsigma}$ is $\check{\xi}_{V,s}$. Furthermore, any solution in a state $\xi_{S,s}$ with $S \neq V$ is infeasible.

This results in the famous recurrence relation of Held and Karp [62] and Bellman [17]

$$C\left(\check{\xi}_{S,i}\right) = \begin{cases} c_{si} & \text{if } |S| = 1 \\ \min_{j \in S \setminus \{i\}} \left\{ C\left(\check{\xi}_{S \setminus \{i\},j}\right) + c_{ji} \right\} & \text{otherwise.} \end{cases}$$

When applying the forward evaluation of this DP algorithm every solution $\check{\xi}_{S,i}$ is expanded to all vertices $j \in V \setminus S$, where the expansion to s is only feasible if $V \setminus S = \{s\}$ to ensure we finish the path in s. This results in algorithm 2.1.

For the DP algorithm for the TSP we have added an extra state variable l to ϕ, since l can take $n = |V|$ possible values and we have that $|\gamma| = 1$ an extra factor n is added to the computational complexity as well as the memory requirement given at the beginning of this chapter. These become $\mathcal{O}(n^2 2^n)$ and $\mathcal{O}(n 2^n)$, respectively. This computational complexity is the best known complexity to

Algorithm 2.1 Forward DP algorithm for the TSP

Input: An instance of the TSP defined by a complete graph $G = (V, E)$, and a distance c_{ij} for edges $e_{ij} \in E$

Output: A sequence ς associated with an optimal route for the TSP

$\check{\xi}_{\emptyset,s} = \varsigma_{\emptyset,s} \wr 0 = \langle \ \rangle$

for $L = 0$ **to** $|V| - 1$ **do**
  for all $S \subset V$ **such that** $|S| = L$ **do**
    $S' = S$
    if $S = \emptyset$ **then**
      $S' = \{s\}$ // Ensure $\check{\xi}_{\emptyset,s}$ is expanded
    for all $i \in S'$ **such that** $\check{\xi}_{S,i} \neq \emptyset$ **do**
      for all $j \in V \setminus S$ **do**
        if $j \neq s$ **or** $V \setminus S = \{s\}$ **then** // Feasibility: ensure we finish in s
          if $\check{\xi}_{S\cup\{j\},j} = \emptyset$ **or** $C(\check{\xi}_{S,i}) + c_{ij} < C(\check{\xi}_{S\cup\{j\},j})$ **then**
            $\check{\xi}_{S\cup\{j\},j} = \check{\xi}_{S,i} \diamond\!\!\Rightarrow j$ // $= \langle \check{\xi}_{S,i}, j \rangle$

return $\check{\xi}_{V,s}$

solve the TSP to optimality. In fact, a constant factor 2 can be removed from the memory requirement as there are at most $\frac{n}{2}$ possible end vertices when $|S| = \frac{1}{2}n$. From the time complexity a constant factor 4 can be removed. Notice that for $|S| = k$ we have k possible end vertices and $n-k$ possible expansions. This results in $\sum_{k=1}^{n} \left(k\,(n-k)\binom{n}{k}\right) + 1 = 2^{n-2}n(n-1) + 1$.

The TSP is often enriched with extra constraints such as time windows in which a location has to be visited, these extensions will be discussed in section 4.3.

2.2 Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is the extension of the TSP to multiple salesmen or vehicles. Given a set of n customer requests R, a set of m vehicles V, with for each vehicle $v_i \in V$ an origin $o_i \in O$ and a destination $d_i \in D$ and a graph $G(R \cup O \cup D, E)$ with a distance c_{ij} for all edges $e_{ij} \in E$. Find routes for each vehicle starting at its origin and finishing at its destination visiting a set of customer requests $R_{v_i} \subseteq R$ such that the total distance is minimized and each request $r \in R$ is only to be visited by a single vehicle $v \in V$, that is $\bigcup_{v_i \in V} R_{v_i} = R$ and $R_{v_i} \cap R_{v_j} = \emptyset$ for $i \neq j$. More information on the VRP can be found in [76,77,116].

Originally the VRP also includes a capacity constraint for each vehicle and a demand at each request. The problem without capacity constraints and all origins and destinations at the same location or depot, demanding the use of m vehicles, is called the Multiple Traveling Salesman Problem or mTSP [15]. To keep the distinction clear in this dissertation all problem variants with multiple routes are called VRPs and problem variants with a single route are called TSPs. Of course a VRP with a single vehicle becomes a TSP but also the mTSP can be transformed into a TSP [57]. This can be done by adding $m - 1$ extra copies of the depot vertex to graph $G(V, E)$ of the TSP with the same distance for all edges connecting to the depot vertices. To enforce the use of m routes the distance between all depot vertices is set to ∞. Now the optimal TSP solution for this new graph will use none of the edges with infinite length creating exactly m routes as there are m copies of the depot which is located at the same location. For now we look at the VRP without extra constraints, the Capacitated Vehicle Routing Problem (CVRP) will be discussed in section 4.3.

To solve the VRP with DP we do essentially the same thing as for the conversion of the mTSP to the TSP, we use vertices for the origin and destination of all vehicles, stitch all routes together and solve it as a single TSP. Combining all routes of a VRP into a single tour is introduced by Funke, Grünert, and Irnich [49] and is called the Giant-Tour Representation (GTR) of a VRP solution. If we order the routes of all m vehicles v_i, $i = 1, \dots, m$ of a VRP solution, then the GTR is a cycle in the graph G where each destination of vehicle i, vertex d_i, is connected to the origin of vehicle $i + 1$, vertex o_{i+1}. Finally the destination of vehicle m, vertex d_m is connected to the origin of vehicle 1, vertex o_1.

In figure 2.1 we see an example of a VRP solution with four vehicles and sixteen customers. The vehicles have their origin and destination at three locations or depots (A, B and C), vehicle 1 *(green)* starts and finishes at depot A, vehicle 2 *(blue)* starts at A and finishes at B, vehicle 3 *(red)* starts and finishes at B and finally vehicle 4 *(yellow)* starts at B and finishes at C. In figure 2.2 we see a GTR of the same VRP solution.

The cycle a GTR forms in G is a TSP solution of graph G. However, not every TSP solution of G is necessarily a GTR of a VRP solution. When we change the distance of all edges of destination to origin vertices to 0, $c_{ij} = 0$ for $i \in D$ and $j \in O$, and all other edges to origin vertices or from destination vertices to ∞, $c_{ij} = \infty$ for $i \notin D$ and $j \in O$ and $c_{ij} = \infty$ for $i \in D$ and $j \notin O$, like the conversion of mTSP to TSP, any TSP solution of G with non-infinite distance will be a GTR of a VRP solution. The distance of this TSP solution will be equal to the distance of the VRP solution as the distance connecting the routes is 0. This converts a VRP into a TSP of size $n + 2m$, leading to a computational complexity of $\mathcal{O}\big((n + 2m)^2\, 2^{n+2m}\big)$.

While adapting the distances in the graph to enforce TSP solutions which are a GTR of a VRP solution is correct, it is often more practical to enforce such constraints by feasibility checks. In the forward calculation of the TSP over $G(R \cup O \cup D, E)$ we select the destination vertex of the last vehicle d_m as our start vertex, so we start our DP algorithm with $\check{\xi}_{\emptyset,d_m}$, and add two feasibility checks to our algorithm. First, any solution $\varsigma_{S,i\wr c}$ can only be expanded to an origin vertex $o \in O$ if and only if the previous node i is a destination vertex, $i \in D$. Second, any solution $\varsigma_{S,i\wr c}$ can only be expanded to a destination vertex $d_k \in D$ if its corresponding origin vertex is already visited $o_k \in S$. This leads to algorithm 2.2.

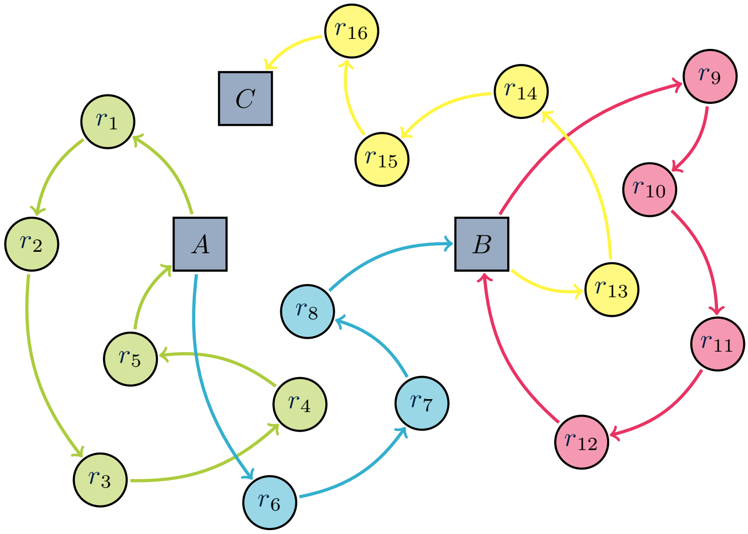


Figure 2.1: *An example of a VRP solution with 4 vehicles*

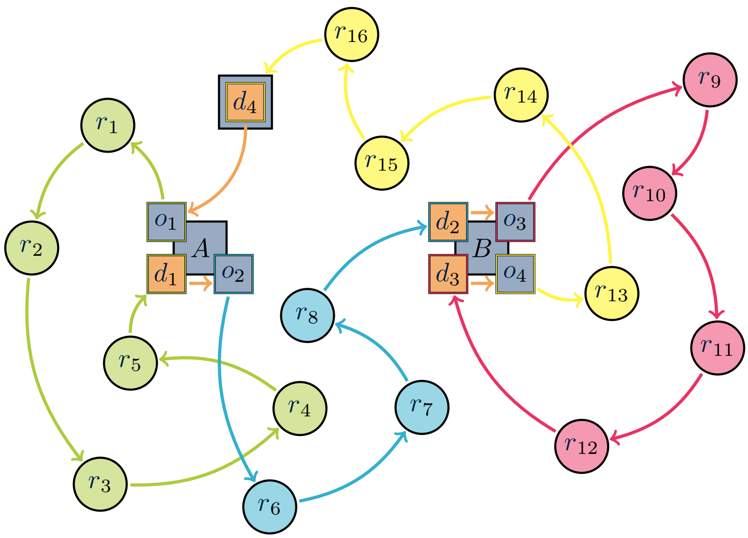


Figure 2.2: *The GTR of the VRP solution in figure 2.1*

2

Algorithm 2.2 Forward DP algorithm for the VRP

Input: An instance of the VRP defined by a set of customer requests R, and a set vehicles V with for each vehicle $v_i \in V$ an origin $o_i \in O$ and destination $d_i \in D$
A Graph $G = (N, E)$, with $N = R \cup O \cup D$ and a distance c_{ij} for all edges $e_{ij} \in E$

Output: A sequence ς associated which is the GTR an optimal solution for the VRP

$\check{\xi}_{\emptyset,d_m} = \varsigma_{\emptyset,d_m} \wr 0 = \langle\ \rangle$

for $L = 0$ **to** $|N| - 1$ **do**
  for all $S \subset N$ **such that** $|S| = L$ **do**
    $S' = S$
    if $S = \emptyset$ **then**
      $S' = \{d_m\}$ // Ensure $\check{\xi}_{\emptyset,d_m}$ is expanded
    for all $i \in S'$ **such that** $\check{\xi}_{S,i} \neq \emptyset$ **do**
      for all $j \in V \setminus S$ **do**
        if $j = d_m$ **and** $V \setminus S \neq \{d_m\}$ **then**
          continue // Feasibility: ensure we finish in d_m
        if $i \in D$ **xor** $j \in O$ **then**
          continue // Feasibility: each origin follows directly a destination
        if $i = d_k \in D$ **and** $o_k \notin S$ **then**
          continue // Feasibility: Allow only destination of current vehicle
        if $\check{\xi}_{S \cup \{j\},j} = \emptyset$ **or** $C(\check{\xi}_{S,i}) + c_{ij} < C(\check{\xi}_{S \cup \{j\},j})$ **then**
          $\check{\xi}_{S \cup \{j\},j} = \check{\xi}_{S,i} \mathbin{\diamond\!\!\Longrightarrow} m$ // $= \langle \check{\xi}_{S,i}, j \rangle$

return $\check{\xi}_{N,d_m}$

Note that, it is also possible to incorporate these feasibility checks into a backward DP algorithm. However, the recurrence relation becomes somewhat tedious

$$C\left(\check{\xi}_{S,i}\right) = \begin{cases} 0 & \text{if } i \in O \text{ and } |S| = 1 \\ \min\limits_{j \in D \cap S} \left\{C\left(\check{\xi}_{S \setminus \{i\},j}\right)\right\} & \text{if } i \in O \text{ and } |S| > 1 \\ \min\limits_{j \in S \setminus (\{i\} \cup D \cup O)} \left\{C\left(\check{\xi}_{S \setminus \{i\},j}\right) + c_{ji}\right\} & \text{if } |S \cap O| = 1 \text{ and } |S| > 1 \\ \min\limits_{j \in S \setminus (\{i\} \cup D \cup \omega(S))} \left\{C\left(\check{\xi}_{S \setminus \{i\},j}\right) + c_{ji}\right\} & \text{otherwise.} \end{cases}$$

Here, $\omega(S)$ is the set of origin vertices o_i in S, $o_i \in O \cap S$, such that the corresponding destination vertices d_i are also in S, $d_i \in D \cap S$.

For the VRP it is possible to fix the order of all vehicles in the GTR, this reduces the computational complexity. In case all vehicles are identical the order of the routes, one for each vehicle, in the GTR has no influence on the solution, so the order of these vehicles can be fixed a priori. For a lot of extensions of the VRP this a priori fixation is possible even if the vehicles are not identical, for example in capacity or depot location. However, when there are relations between vehicles, for example a vehicle v_i may only depart after the arrival of vehicle v_j, not every order of vehicles can be allowed in the GTR as the arrival time of vehicle v_j, at vertex d_j, must be known before the feasibility of any extension to o_i can be performed. In fact, as long as the relations between the vehicles are known beforehand and are not circular, a proper order of the vehicles in the GTR can be found.

To fix the order of the vehicles in the GTR we add a constraint that vertex d_i has to be followed directly by vertex o_{i+1}, assuming the GTR is ordered according to the index of the vehicles. Since these two vertices are adjacent in any solution, we can merge vertices d_i and o_{i+1} into a single vertex d_i using the incoming edges of d_i and the outgoing edges of o_{i+1}, thereby removing the m vertices of O from the graph G. Now the two extra feasibility checks can be removed. However, to ensure the serial precedence relation between the merged vertices D according to the fixed vehicle order, a new feasibility check must be added. That is, any solution $\varsigma_{S,i \wr c}$ can only be expanded to vertex $d_k \in D$ if $\bigcup_{i=1}^{k-1} d_i \subseteq S$.

Since we removed m vertices, the computational complexity is already reduced to $\mathcal{O}\big((n+m)^2\, 2^{n+m}\big)$, but the serial precedence relation of length m also reduces the complexity by a factor $\frac{2^m}{m+1}$, see section 4.2.1. This results in a computational complexity of $\mathcal{O}\big((n+m)^2\, m 2^n\big)$ for the DP algorithm for the VRP. Note that the complexity is $\mathcal{O}^*(2^n)$ so it depends heavily on the number of request and far less on the number of vehicles of the instance. Here, $\mathcal{O}^*()$ is defined as $\mathcal{O}()$ by omitting any polynomials, see [122].

2.3 Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) is similar to the problem described in section 1.2.3. However, in the basic Job Shop Scheduling Problem ($J||C_{\max}$ [see 58]) we do not look at deadlines or weights associated with the tardiness. In a JSSP we have N jobs that have to be processed on M dedicated machines. Each job has a set of operations that should be processed following a specific order, each operation must be processed by a specific machine. The time each job requires on each machine depends on the job and on the machine and it is assumed to be known in advance. A machine can process only one job at the time and no job can be processed simultaneously on two or more machines. Preemption is not allowed, meaning that when a machine starts processing a job it should finish operating on that job before starting on another job. Note that, different machines can run operations of different jobs in parallel. The goal is to schedule the jobs so as to minimize the makespan, which is the maximum of their completion times. More information on the JSSP can be found in [30,22,96].

In the rest of this dissertation we make the extra assumption that each job has exactly one operation to be processed on each machine. This assumption, made to simplify the notation, is not used by the DP algorithm which can in fact solve the general JSSP where it is allowed to have an arbitrary number of operations for each machine. This fact is used by the DP algorithm described in section 6.3 where maintenances are modeled as a special type of jobs. This assumption *is* used in the complexity analyses where the number of machines is used as the number of operations for all jobs.

Let $\mathcal{J} = \{j_1, j_2, \ldots, j_N\}$ denote the set of N jobs and $\mathcal{M} = \{m_1, m_2, \ldots, m_M\}$ the set of M machines. Each job consists of M operations each of which is associated with a specific machine, $p_{mj} \in \mathbb{N}$ is the processing time of the operation of job $j \in \mathcal{J}$ on machine $m \in \mathcal{M}$. The sequence of operations defines for each job the order in which the machines have to be visited and is denoted by $\pi_j(1), \ldots, \pi_j(M)$, that is, for job $j \in \mathcal{J}$, $\pi_j(i)$ is the i-th machine that job j has to visit. $\mathcal{O} = \{o_1, o_2, \ldots, o_{N\times M}\}$ is the set of operations. The first N operations refer to the first operation of each job (in the order of the jobs), operations $o_{N+1}, \ldots, o_{2N}$ concern the second operation for each of the N jobs, and so on. For an operation $o \in \mathcal{O}$ we denote by $j(o)$ and $m(o)$ the corresponding job and machine, respectively. Note that $j(o_i) = i \bmod N$. We denote by $p(o)$ the processing time of operation $o \in \mathcal{O}$. Note that $p(o) = p_{m(o)j(o)}$. The goal is to find a feasible schedule that minimizes the makespan.

Definition 2.1
A schedule is a function $\psi : \mathcal{O} \to \mathbb{N}_0$ such that for each operation $o \in \mathcal{O}, \psi(o)$ gives the starting time of operation o. A schedule ψ is said to be feasible if:

1. *$\psi(o) \geq 0$ for each $o \in \mathcal{O}$;*
2. *For all $o_k, o_l \in \mathcal{O}$ such that $j(o_k) = j(o_l)$ and $k < l$ holds that $\psi(o_k) + p(o_k) \leq \psi(o_l)$;*
3. *For all $o_k, o_l \in \mathcal{O}$ such that $k \neq l$ and $m(o_k) = m(o_l)$ holds that $\psi(o_k) + p(o_k) \leq \psi(o_l)$ or $\psi(o_l) + p(o_l) \leq \psi(o_k)$.* □

Similarly we define a partial schedule for a set of operations $S \subset \mathcal{O}$, which can only be feasible if for all operations in S all preceding operations of the same job are also in S. The makespan $C_{\max}$ of a schedule ψ is $C_{\max}(\psi) = \max_{o\in\mathcal{O}} \{C_o\}$, where $C_o = \psi(o) + p(o)$ is the finish time of operation o. The makespan can similarly be defined for a partial schedule with $\max_{o\in S}$.

To use DP for the JSSP we want to be able to schedule each operation separately, in order to achieve this we create a DP algorithm over all operations $\mathcal{O}$. A solution in the DP algorithm will be represented by a sequence of operations similar to the TSP and VRP. First we limit the type of schedule we are interested in by limiting the search to no-idle schedules. A no-idle schedule is a schedule where no operation can be feasibly scheduled at an earlier time without changing the order of operations on any machine. Any operation in a no-idle schedule starts either at time 0, directly follows another operation on the same machine *or* directly follows the predecessor of the same job. To be able to represent a solution by a sequence of operations such that these operations $\mathcal{O}$ become the nodes of the DP algorithm, we need to find a way to correspond each schedule with such a sequence and vice-versa. So we want to have a bijection between no-idle schedules and feasible sequences. Any well defined ordering of operations according to a schedule can create such bijection. A possible ordering could be all operations of the first machine in the order of that machine followed by the operations on the second machine, etc. However, to be able to determine the schedule time of an operation at the moment of expansion of the sequence it is important that two orders of the operations are preserved in the sequence. First for each job i its operations ($j(o) = i$) must be ordered according to the order they have to be executed. Note this order is the same order as the index of the operation for the job are ordered, for job i its operations $o_i, o_{i+N}, o_{i+2N}, \ldots, o_{i+(M-1)N}$ have to be executed in that order. Furthermore, for each machine the operations should be ordered according to the operation order on that machine according to schedule ψ. An example of a sequence preserving both orders is the sequence which is ordered according to the starting time $\psi(o)$ of each operation o. These two properties are important as this allows us to calculate the (no-idle) starting time of an operation o as extension of a sequence using only the starting times of the operations preceding it in the sequence.

Each sequence which respects the first ordering automatically defines a no-idle schedule which is a feasible solution to the Job Shop Scheduling Problem. However, multiple sequences can define the same no-idle schedule.

When we define the sequence ς of a schedule ψ to be the operations only ordered according to the starting time of the operations in schedule ψ, we have no one-to-one correspondence between the schedules and sequences. A simple

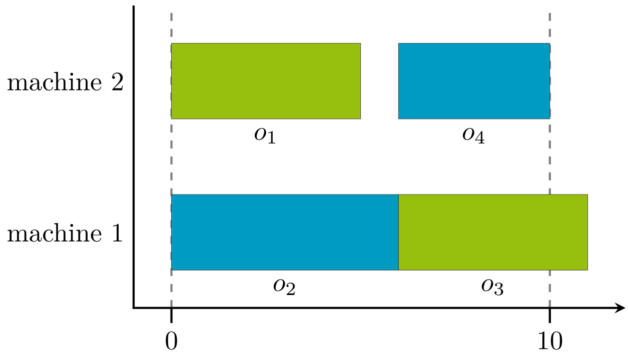


Figure 2.3: *A simple JSSP schedule with two jobs, green and blue*

example with two jobs; for the schedule in figure 2.3 the sequences $o_1 o_2 o_3 o_4$, $o_1 o_2 o_4 o_3$, $o_2 o_1 o_3 o_4$ and $o_2 o_1 o_4 o_3$ are all sorted by starting time of the operations, and thereby preserving the ordering on each machine as well as the order of the operations of each job. Furthermore, a sequence can be defined by several schedules, for example the schedule of figure 2.3 with operation o_4 delayed defines the same four sequences. To define a unique sequence for every schedule we need the limitations to no-idle schedules where there is no extra idle time, that is every operation is scheduled directly after the previous operation on the same machine or directly after the previous operation of the same job. Naturally every feasible schedule of a JSSP can be transformed in a no-idle schedule by advancing all operations that have extra idle time.

For the DP algorithm we sort the operations of a no-idle schedule according to the finish time of each operation, where we use the machine number as a tie-breaker.

Proposition 2.2
For every feasible no-idle (partial) schedule for the Job Shop Scheduling Problem there is one and only one (partial) sequence of operations defining the schedule with the operations sorted such that:

- *The completion times of the operations along the sequence are non decreasing*
- *The machine numbers $m(o)$ are increasing for two consecutive operations with equal completion time.* □

2

Proof Consider a feasible no-idle schedule for the Job Shop Scheduling Problem. A sequence of operations featuring the conditions is obtained by sorting the operations in non-decreasing order of their completion times $(\psi(o) + p(o))$ and for those that have an equal completion time, by sorting them in increasing order of the machine associated with them. The total lexicographic order imposed on this sequence guarantees uniqueness, since no two operations with the same completion time are scheduled on the same machine. ■

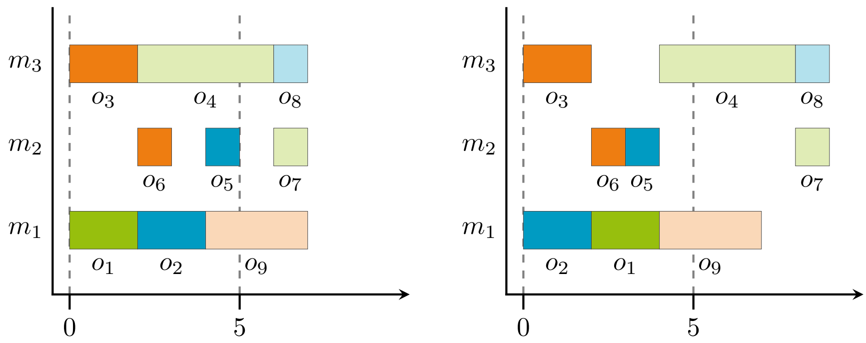
Notice that for every sequence which preserves the order of operations for each job one feasible no-idle schedule can be found, by scheduling the operations as soon as possible after the operations already present on its machine and after all preceding operations of the same job. However, such a sequence is not necessarily ordered corresponding to proposition 2.2, this leads us to the following definition.

Definition 2.3
A (partial) sequence that defines a feasible (partial) schedule is called ordered when it is ordered according to proposition 2.2. Otherwise it is called unordered. □

For example, of the four sequences $o_1 o_2 o_3 o_4$, $o_1 o_2 o_4 o_3$, $o_2 o_1 o_3 o_4$ and $o_2 o_1 o_4 o_3$ that lead to the schedule of figure 2.3, only the sequence $o_1 o_2 o_4 o_3$ is ordered.

If we limit the sequences to ordered sequences we have a bijection between ordered sequences and no-idle schedules. Since we have a no-idle variant for each schedule with equal or possibly lower $C_{\max}$, any optimal solution can be represented as a no-idle schedule with the same $C_{\max}$. When we limit the search in the DP algorithm to ordered sequences we do not disregard any no-idle optimal solutions. The choice of sorting on the finish times of the operations will become important later as this ensures that the finish time of the last operation in a (partial) ordered sequence defines the $C_{\max}$ of that sequence.

Now we want to find the sequence corresponding to an optimal schedule of a JSSP using DP over the set of all operations $\mathcal{O}$. For each (partial) schedule, and thereby for the optimal schedule, we have a corresponding ordered sequence, so during the DP algorithm we only consider solutions which correspond to ordered sequences as feasible. Still a simple example is enough to show that a state definition of $\xi_{S \wr C_{\max}}$ will not suffice. In figure 2.4 we see two solutions of the



a: *An optimal solution* **b:** *A "dominating" partial solution*

Figure 2.4: *Two solutions of the same JSSP instance*

same JSSP, the dark colors refer to the operations in a partial schedule, while the lighter colors denote its completions. The partial solution $o_1 o_3 o_6 o_2 o_5$ of the optimal solution in figure 2.4a is with this state definition dominated by the partial solution $o_2 o_3 o_6 o_1 o_5$ of figure 2.4b, since its $C_{\max}$ is lower than the partial solution of the optimal solution. So we need extra state variables to keep the principle of optimality.

As state variables we are going to use for each job the earliest possible finish time of the first unscheduled operation of that job when it is scheduled in an ordered sequence. We will first define this time for each job and then show that using these as state variables the optimality principle holds.

Denote by $\varepsilon(S) \subseteq \mathcal{O}$ the set of operations that consist of the first operation of each job that is not in S, and denote by $\lambda(S) \subseteq S$ the set consisting of the last operation in S for each job. Note that $|\varepsilon(S)| \leq N$ and $|\lambda(S)| \leq N$, since there is at most one such operation per job. Furthermore, there are $N - |\varepsilon(S)|$ jobs that have all operations in S, and similarly there are $N - |\lambda(S)|$ jobs with no operation in S. Any solution ς_S can only be feasibly expanded to an operation $o \in \varepsilon(S)$ as any other expansion will result in the sequence corresponding to ς_S to become unordered.

For each solution ς_S and each operation $o \in \varepsilon(S)$ define $\psi(\varsigma_S, o)$ as the starting time of o in the schedule of the expansion $\varsigma_S \diamond\!\!\Rightarrow o$ even if adding o to the sequence ς_S leads to an unordered sequence. Let for a solution ς_S the set $\eta(\varsigma_S)$ be the set of all possible expansions $\varsigma_S \diamond\!\!\Rightarrow o$ where the sequence of this expansion is ordered, thereby rendering the expansion — within the DP algorithm — feasible, naturally $\eta(\varsigma_S) \subseteq \varepsilon(S)$. Let $\Lambda(\varsigma_S)$ be the last operation in the sequence of ς_S, now $\eta(\varsigma_S)$ is defined by

$$\eta(\varsigma_S) = \left\{ o \in \varepsilon(S) \;\middle|\; \begin{array}{l} \psi\left(\varsigma_S, o\right) + p(o) > C_{\max}(\varsigma_S) \text{ or} \\ \psi\left(\varsigma_S, o\right) + p(o) = C_{\max}(\varsigma_S) \wedge m(o) > m\left(\Lambda\left(\varsigma_S\right)\right) \end{array} \right\}.$$

For the first case, it follows directly that the sequence of the expansion with o is ordered. For the second case, note that as ς_S corresponds to an ordered sequence and that $m\left(\Lambda\left(\varsigma_S\right)\right)$ is the machine with the highest machine number within the machines with eh highest completion time in the schedule of ς_S. Although the expansion with operation o has the same completion time, o has a higher machine number so the expansion with o is also ordered.

To create a state definition in such a way that we can be sure that certain partial solutions are in fact dominated we define an *aptitude* value for each operation in $o \in \varepsilon(S)$. This will eventually allow us to schedule all operations of a dominated solution as a completion of the dominating solution.

Definition 2.4
We define an 'aptitude' value for a solution ς_S and each operation $o \in \varepsilon(S)$ as

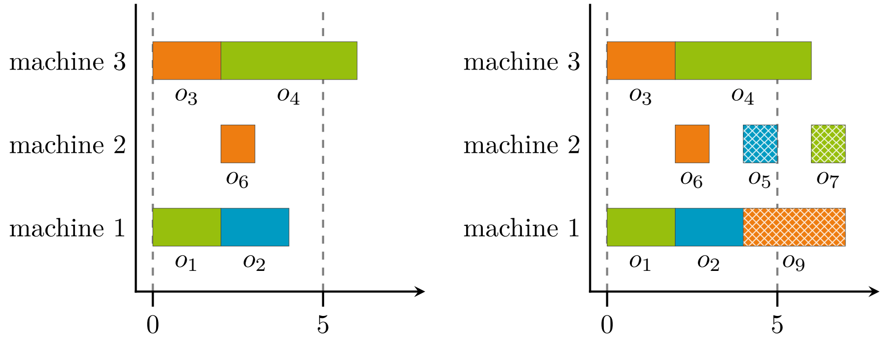
$$\alpha(\varsigma_S, o) = \begin{cases} \psi(\varsigma_S, o) + p(o), & \text{if } o \in \eta(S) \\ C_{\max}(\varsigma_S) + p(o), & \text{otherwise.} \end{cases}$$

□

This aptitude value $\alpha(\varsigma_S, o)$ represents the earliest completion time of operation o in any ordered completion $\varsigma_{\mathcal{O}}$ of ς_S. For the first case, when $o \in \eta(S)$, ς_S can directly feasible be expanded with o, $\alpha(\varsigma_S, o)$ is directly defined as the completion time of o in the expansion $\varsigma_S \Diamond\!\!\Rightarrow o$. For the second case, when $o \notin \eta(S)$, the sequence of the expansion $\varsigma_S \Diamond\!\!\Rightarrow o$ is not ordered. This means that neither the completion time of machine $m(o)$ nor the completion time of the operation preceding o in the job $j(o)$ is limiting the start time of o in $\varsigma_{\mathcal{O}}$ such that is able to be completed at or before $C_{\max}(\varsigma_S)$. Since $\varsigma_{\mathcal{O}}$ is ordered, there has to be another operation $o' \notin S$ with $m(o') = m(o)$ which precedes o in the sequence of any completion $\varsigma_{\mathcal{O}}$. For completion time of o' in the schedule $\psi_{\varsigma_{\mathcal{O}}}$ we have that $\psi_{\varsigma_{\mathcal{O}}}(o') + p(o') \geq C_{\max}(\varsigma_S)$, now the earliest completion time $C_{\max}(\varsigma_S) + p(o)$ for o follows directly.

In order to clarify the previous concepts, we consider the instance of the JSSP that was introduced in figure 2.4. In particular, for this instance, consider a partial solution ς_S with sequence $o_1\, o_3\, o_6\, o_2\, o_4$, which leads to the schedule depicted in figure 2.5a. In this case we have $C_{\max}(\varsigma_S) = 6$, $S = \{o_1, o_2, o_3, o_4, o_6\}$, $\varepsilon(S) = \{o_5, o_7, o_9\}$, $\lambda(S) = \{o_2, o_4, o_6\}$ and $\eta(\varsigma_S) = \{o_7, o_9\}$. Taking into account that $p(o_5) = 1$, $p(o_7) = 1$ and $p(o_9) = 3$ we obtain (see figure 2.5b): $\psi(\varsigma_S, o_5) = 4$, $\psi(\varsigma_S, o_7) = 6$, $\psi(\varsigma_S, o_9) = 4$, $\alpha(\varsigma_S, o_5) = 7$, $\alpha(\varsigma_S, o_7) = 7$, $\alpha(\varsigma_S, o_9) = 7$. Note that the expansion $\varsigma_S \Diamond\!\!\Rightarrow o_5$ is not regarded as feasible within the DP algorithm as $o_5 \notin \eta(\varsigma_S)$ and therefore will not lead to an ordered expansion.

Let $\vec{\alpha}(\varsigma_S)$ be the array of $\alpha(\varsigma_S, o)$ for all $o \in \varepsilon(S)$ ordered according to $j(o)$, and let $\vec{\alpha}$ be the single element $C_{\max}$ when $\varepsilon(S) = \emptyset$ (thus $S = \mathcal{O}$). Notice that $|\vec{\alpha}(\varsigma_S)| = |\varepsilon(S)| \leq N$ and that for each job at most one operation is represented in $\vec{\alpha}$, this ensures that for a single S the same operations are represented in the



a: *Scheduled defined by* ς_S **b:** *Possible schedules defined by* $\varsigma_S \diamond\!\!\Rightarrow o$

Figure 2.5: *Illustration of the values* $\psi(\varsigma_S, o)$ *and* $\alpha(\varsigma_S, o)$ *for* $o \in \varepsilon(S)$

same order in $\vec{\alpha}$. Furthermore, let $\vec{\eta}(\varsigma_S)$ be the array of $\eta(\varsigma_S, o)$ representing the same operations $o \in \varepsilon(S)$ in the same ordering as $\vec{\alpha}$, where $\eta(\varsigma_S, o)$ is defined as

$$\eta(\varsigma_S, o) = \begin{cases} 1, & \text{if } o \in \eta(\varsigma_S) \\ 0, & \text{otherwise.} \end{cases}$$

Note that each job is represented in $\vec{\alpha}$ and $\vec{\eta}$ at a distinct location regardless of the current operation in $\varepsilon(S)$, this location is removed when all operations of a job are scheduled.

The aptitudes $\vec{\alpha}(\varsigma_S)$ gives us the completion times of the expansion of ς_S with any operation in $\varepsilon(S)$, and the array $\vec{\eta}(\varsigma_S)$ specifies whether the expansion with such an operation is ordered. These to vectors give us the instrument to define a correct domination and a state definition which provides the principle of optimality. Before we create this state definition we show some important properties of $\vec{\alpha}$ and $\vec{\eta}$. For the array $\vec{\alpha}$ we define $\geq$ in a similar way as for γ, using $\leq$ for each element-wise compare. That is $\vec{\alpha}(\varsigma_S^1) \geq \vec{\alpha}(\varsigma_S^2)$ when $\alpha(\varsigma_S^1, o) \leq \alpha(\varsigma_S^2, o)$ for all $o \in \varepsilon(S)$.

Proposition 2.5
When $\vec{\alpha}(\varsigma_S^1) \geq \vec{\alpha}(\varsigma_S^2)$ *any operation in* $\mathcal{O} \setminus S$ *of any ordered completion* $\varsigma_{\mathcal{O}}^2$ *of* ς_S^2 *can be scheduled at the same time in the schedule of* ς_S^1*. This leads to a feasible possibly no-idle schedule of the JSSP with a makespan of* $C_{\max}(\varsigma_{\mathcal{O}}^2)$*.* $\square$

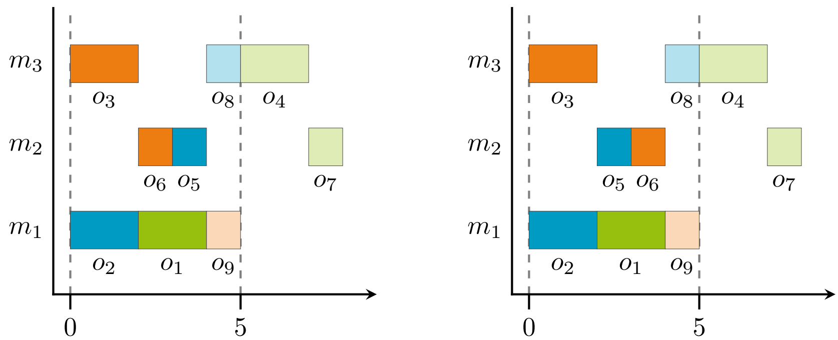
Proof For all operations $o \in \varepsilon(S)$ we have that $\alpha(\varsigma_S^1, o) \leq \alpha(\varsigma_S^2, o)$, so that any expansion of ς_S^2 can be scheduled either at the same time as or earlier than the expansion of ς_S^1. Thus, all operations $o \in \varepsilon(S)$ can be scheduled at the same time in ς_S^1 as they are scheduled in $\varsigma_{\mathcal{O}}^2$. Note that for the completion time of such an operation the following holds:

$$\psi(\varsigma_{\mathcal{O}}^2, o) + p(o) \geq \alpha(\varsigma_S^2, o) \geq \alpha(\varsigma_S^1, o) \geq C_{\max}(\varsigma_S^1).$$

Any other operation can as well be scheduled at the same time as they can only be scheduled when any preceding operation of the same job is finished. Since this includes an operation in $\varepsilon(S)$ each of the operations in $\mathcal{O} \setminus (S \cup \varepsilon(S))$ has a predecessor in $\varepsilon(S)$ and can thereby only start at a time later than $C_{\max}(\varsigma_S^1)$. ∎

Note that the last argument of the preceding proof shows why we need the ordering on finish times of the sequences.

We use a small example to clarify this idea. When we take a look at figure 2.6 the partial solution $o_2 o_3 o_6 o_1 o_5$ in figure 2.6a is dominated by the partial solution $o_2 o_3 o_5 o_1 o_6$ in figure 2.6b, since the aptitude values for o_4 and o_9 are equal and the aptitude value for o_8 is lower for the partial solution in figure 2.6b. We can also see that the completion $o_9 o_8 o_4 o_7$ of $o_2 o_3 o_6 o_1 o_5$ in figure 2.6a can be added in the schedule of $o_2 o_3 o_5 o_1 o_6$ see figure 2.6b.



a: *Schedule of $o_2 o_3 o_6 o_1 o_5$ with expansion $o_8 o_9 o_4 o_7$*

b: *Schedule of $o_2 o_3 o_5 o_1 o_6$ where the expansion of figure 2.6a is added*

Figure 2.6: *The completion of one schedule is added in another schedule*

However, such a schedule constructed by completing a partial schedule with the completion of another partial schedule can be no-idle. In figure 2.6b operation o_8 is idle and can be moved forward (together with operations o_4 and o_7). This will possibly also result in a change in the order of the operations. In figure 2.6b the completion would become $o_8 o_9 o_4 o_7$ instead of the original $o_9 o_8 o_4 o_7$.

In fact, it could be possible that creating a no-idle schedule from such a schedule would move one or more operations of the completion of the dominated schedule before the last operation of the dominating schedule according to the order of the resulting sequence. In figure 2.6 this would be the case if operation o_1 would have length 3, see figure 2.7. When this is the case, the completion of the dominated solution cannot be scheduled as an ordered completion of the dominating solution. This is where $\vec{\eta}$ is needed, we see that in that case the values of η would differ for o_8 for the two schedules in figure 2.6.

Proposition 2.6
When $\vec{\eta}(\varsigma_S^1) = \vec{\eta}(\varsigma_S^2)$ for the two solutions in proposition 2.5 the sequence of the no-idle schedule created from the schedule starting with ς_S^1 with a completion of ς_S^2 added starts with the sequence of ς_S^1. □

Proof Proposition 2.5 ensures that the schedule is feasible. Any operation $o \in \varepsilon(S) \setminus \eta(\varsigma_S^1)$ ($\eta(\varsigma_S^1, o) = 0$) cannot be scheduled as an ordered expansion of ς_S^1. For o to be scheduled such that the resulting completion starts with ς_S^1, another operation should be scheduled in the completion on the same machine $m(o)$ before o can be scheduled in any ordered sequence. The fact that also $\eta(\varsigma_S^2, o) = 0$ ensures that such an operation is planned in any ordered completion of ς_S^2. ∎

When we add $\vec{\alpha}$ and $\vec{\eta}$ to the state definition we can create a DP algorithm for which the principle of optimality holds. We add $\vec{\eta}$ to the fixed variables of the state definition leading to $\phi = (S, \vec{\eta})$, and use $\vec{\alpha}$ as the comparable part of the state definition, thus $\gamma = \vec{\alpha}$. For $\vec{\alpha}$ we define $\geq$ similar to γ using $\leq$ for each element-wise compare. So for two solutions $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}$ and $\varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ we have $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}} \geq \varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ when all values of $\vec{\alpha}$ are less or equal to their corresponding values in $\vec{\alpha}'$ ($\vec{\alpha} \geq \vec{\alpha}'$).

Proposition 2.7
For the state definition $\xi_{S,\vec{\eta}\wr\vec{\alpha}}$ the optimality principle holds. □

Proof For the optimality principle to hold, first all state variables of an expansion must follow directly from the expanded solution and the choice of the expansion. When we have an expansion $\varsigma' = \varsigma_{S,\vec{\eta}\wr\vec{\alpha}} \diamond\!\!\Rightarrow o_i$, this expansion is only feasible if $o_i \in \eta(S) \subseteq \varepsilon(S)$. This can be directly deduced from the value of $\vec{\eta}$ for the corresponding job $j(o_i)$. When the expansion is feasible the completion time of o_i in ς' is $\psi(\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}, o_i) + p(o_i) = \alpha(\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}, o_i)$ and is equal to the value of $\vec{\alpha}$ for $j(o_i)$. Note that $C_{\max}(\varsigma') = \alpha(\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}, o_i)$, and that $\varepsilon(S \cup \{o_i\}) = \varepsilon(S) \setminus \{o_i\} \cup \{o_{i+N}\}$, where o_{i+N} is only added if o_i is not the last operation of job $j(o_i)$. So the operations represented in $\vec{\alpha}$ and $\vec{\eta}$ stay the same, except o_{i+N} is represented instead of o_i (or the representation of $j(o_i)$ is removed when o_i is the last operation of this job).

For the expansion $\varsigma'_{S',\vec{\eta}'\wr\vec{\alpha}'}$ we have that $S' = S \cup \{o_i\}$, $\vec{\eta}' = \vec{\eta}$ but its value corresponding to o_{i+N} ($\eta(\varsigma', o_{i+N})$) is set to 1, and the value is set to 0 for all operations in $o \in \varepsilon(S')$ that cannot be expanded as an ordered sequence anymore. That is if $\alpha(\varsigma, o) < C_{\max}(\varsigma')$ or $\alpha(\varsigma, o) = C_{\max}(\varsigma')$ *and* $m(o) \leq m(o_i)$. The values for $\vec{\alpha}'$ are equal to $\vec{\alpha}$ except for all operations $o \in \varepsilon(S')$ where $\eta(\varsigma', o) = 0$ or $m(o) = m(o_i)$ and for o_{i+N} these are set to $C_{\max}(\varsigma') + p(o)$. Note that all these values are available from the previous state or are deduced earlier. So all new state variables of an expansion follow directly from the previous state and the choice (operation) of the expansion.

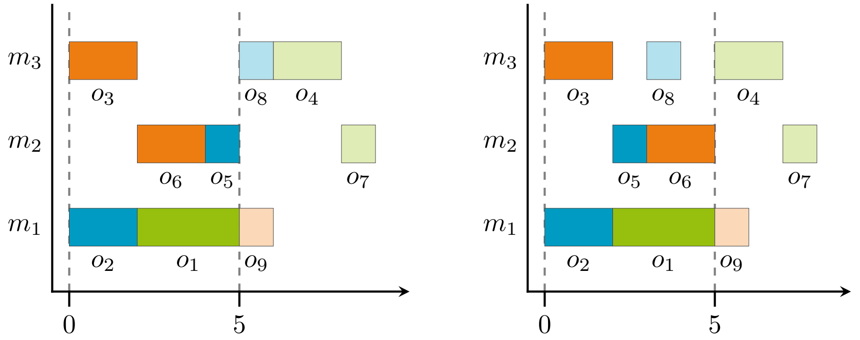
Finally, for the optimality principle to hold, when we have two solutions $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}$ and $\varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ in the same state $\xi_{S,\vec{\eta}}$ and $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}} \geq \varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ ($\vec{\alpha} \geq \vec{\alpha}'$), all feasible expansions and completions of ς' must be dominated by the same expansion made to ς. Since $\vec{\eta}$ is equal for both solutions, exactly the same operations give a feasible expansion. For the completion time $\psi(\varsigma', o)$ of any expansion with o of ς' we have that $\psi(\varsigma, o) \leq \psi(\varsigma', o)$, as $\vec{\alpha} \geq \vec{\alpha}'$. We have not necessarily that $\vec{\eta}^* = \vec{\eta}'^*$. So for the expansions ς^* and ς'^* of ς and ς', respectively, we have that $\vec{\alpha}^* \geq \vec{\alpha}'^*$ except for the operations where $\eta^*(\varsigma^*, o) \neq \vec{\eta}'^*(\varsigma'^*, o)$.

So the expansion of ς may not dominate the expansion of ς' directly. When we look at a completion $\varsigma'_{\mho}$ of ς', the operations of this completion can be scheduled at exactly the same times after ς, leading to a feasible (possibly idle) schedule ψ. This schedule can be converted to a no-idle schedule ψ^* with an equal or lower completion time. The ordered sequence corresponding to ψ^* starts with the sequence of ς as for ς and ς' had equal $\vec{\eta}$, for all operations $o \in \varepsilon(S) \setminus \eta(\varsigma)$ another operation has to be scheduled before o on the machine $m(o)$ before o can be scheduled in the completion of ς'. This prevents the completion time of any operation to advance before $C_{\max}(\varsigma)$ in the conversion from ψ to ψ^*. So any completion of ς' is dominated by a completion of ς. ∎

Since proposition 2.5 does not use the values of $\vec{\eta}$ this result is independent of the fact that $\vec{\eta} = \vec{\eta}'$ as used in propositions 2.6 and 2.7. When we have two solutions $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}$ and $\varsigma'_{S,\vec{\eta}'\wr\vec{\alpha}'}$ for which we have $\vec{\alpha} \geq \vec{\alpha}'$ and $\vec{\eta} \neq \vec{\eta}'$, according to proposition 2.5 we have that the operations of any completion $\varsigma'_{\mho}$ of ς' can be scheduled at the same times after ς leading again to a schedule ψ. When we convert this again to a no-idle schedule ψ^* the corresponding sequence does not have to start with the same sequence as ς, since it may be the case that in the sequence corresponding to ψ^* operations of the completion are scheduled before operations from ς. However, this ensures a solution with equal or better makespan as $\varsigma'_{\mho}$ does exist.

This suggests that we can leave $\vec{\eta}$ out of the state definition, as used in proposition 2.7, although ς' is not directly dominated by ς, ς' is dominated by some solution somewhere in the state space. ς' is not directly dominated by ς because it may be possible that for some completions of ς, that can be made by adding the best possible completions of ς' to ς, the sequences are *unordered*. The ordered sequences belonging to such a schedule do not start with ς, as is showed in figure 2.7.

Because $\vec{\eta}$ is used in the calculation of the state variables, we have to prove that there is at least one optimal solution that is not dominated in any stage. To not remove $\vec{\eta}$ entirely from the state definition we introduce a new set of state variables next to ϕ and γ. We define β as the set of bookkeeping variables needed in the calculation of all state variables but not used to divide the state space into states (ϕ) or compare solutions within a state (γ). So we define $\beta = (\vec{\eta})$. In the state definition we divide the variables of ϕ and β with $\wr$. The state definition for the DP algorithm for the JSSP now becomes $\xi_{S\wr\vec{\alpha}\wr\vec{\eta}}$. Theoretically this could reduce the complexity of the algorithm by a factor 2^N, as the theoretical number of states is reduced for all possible values of $\vec{\eta}$ to 1. However, in practice this factor will be less, since typically not all possible states are created. Further complexity analysis can be found at the end of this section. When bookkeeping variables β are used, an extra proof is needed to prove that an optimal solution is found. Before we can prove this we first have to make a couple of observations.



a: *Schedule of* $o_2 o_3 o_6 o_1 o_5$ *with expansion* $o_9 o_8 o_4 o_7$

b: *Schedule of* $o_2 o_3 o_5 o_1 o_6$ *where the expansion of figure 2.7a is added no-idle*

Figure 2.7: *Operation* o_8 *of a completion is scheduled before the last operation of the dominating sequence*

The state definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$ allows for indirect domination, even of optimal solutions. Let $\varsigma^1_{\mathcal{O}}$ be a complete solution, and let $\varsigma^1_{\mathcal{O}}$ be a completion of a partial solution ς^1 ($\varsigma^1_{S \wr \vec{\alpha}^1 \wr \vec{\eta}^1}$) that is dominated during the DP algorithm. Then there is a solution ς^2 ($\varsigma^2_{S \wr \vec{\alpha}^2 \wr \vec{\eta}^2}$) with the same operations and $\vec{\alpha}^2 \geq \vec{\alpha}^1$. According to proposition 2.5 we can add the operations $\mathcal{O} \setminus S$ of any completion of ς^1 (also the one leading to $\varsigma^1_{\mathcal{O}}$) to the schedule of ς^2 at exactly the same times as they were scheduled in the extension of ς^1 (to possibly $\varsigma^1_{\mathcal{O}}$). From such, possibly idle, schedule — starting with ς^2 and adding the operations $\mathcal{O} \setminus S$ at the time they are scheduled in ς^1 — we create a no-idle schedule ψ^2, by advancing all operations as much as possible. Obviously, such schedule represented by sequence $\varsigma^2_{\mathcal{O}}$, will have a makespan equal or lower than the original solution $\varsigma^1_{\mathcal{O}}$ which is a completion of ς^1. We say $\varsigma^2_{\mathcal{O}}$ is the solution *welded* from the partial solution ς^2 and the completion of ς^1 to $\varsigma^1_{\mathcal{O}}$.

We can categorize all completions $\varsigma^1_{\mathcal{O}}$ of ς^1 with respect to the domination by ς_2 into two cases based in the solution $\varsigma^2_{\mathcal{O}}$ welded from ς_2 and the completion of ς_1 to $\varsigma^1_{\mathcal{O}}$:

I. The welded sequence $\varsigma^2_{\mathcal{O}}$ starts with the sequence represented by the partial solution ς^2. This implies that the completion of partial solution ς^1 to solution $\varsigma^1_{\mathcal{O}}$ can be scheduled as an ordered, no-idle, completion of partial solution ς^2. For all completions of this type we have a partial solution, namely ς^2, which can be expanded to a solution with equal or lower makespan. We call this *direct domination.*

II. The welded sequence $\varsigma^2_{\mathcal{O}}$ does not start with the sequence represented by the partial solution ς^2. This implies that at least one operation $o \in \mathcal{O} \setminus S$ in schedule ψ^2 is advanced so that this operation occurs in the ordered sequence before the last operation $\Lambda(\varsigma^2)$ of the sequence represented by ς^2.

This implies that $\alpha(\varsigma^2, o) = C_{\max}(\varsigma^2) + p(o)$ as otherwise the expansion of o could be done in an ordered way. This actually ensures that for each completion of ς^1 there can be another solution welded from ς^2 and this completion with equal or lower makespan. We call this *indirect domination*.

When we have indirect domination (case II.) we can deduce some special properties.

Proposition 2.8
If we have indirect domination between ς^1 and ς^2 as described in case II., there is at least an operation $o \in \mathcal{O} \setminus S$ that is scheduled in the welded solution $\varsigma^2_{\mathcal{O}}$ such that o is finished in $\varsigma^2_{\mathcal{O}}$ before it is scheduled to start in $\varsigma^1_{\mathcal{O}}$. □

Proof Since we have indirect domination, there is at least one operation o that is scheduled in $\varsigma^2_{\mathcal{O}}$ before $\Lambda(\varsigma^2)$. As operation o could not be scheduled as expansion of ς^2 leading to an ordered schedule we have the following

$$\psi(\varsigma^1_{\mathcal{O}}, o) + p(o) \geq \alpha(\varsigma^1_S, o) \geq \alpha(\varsigma^2_S, o) = C_{\max}(\varsigma^2_S) + p(o).$$

From this we can conclude that

$$\psi(\varsigma^1_{\mathcal{O}}, o) \geq C_{\max}(\varsigma^2_S) = \psi(\varsigma^2_{\mathcal{O}}, \Lambda(\varsigma^2)) + p(\Lambda(\varsigma^2)) \geq \psi(\varsigma^2_{\mathcal{O}}, o) + p(o).$$ ■

Corollary 2.9
Operation o of proposition 2.8 can be scheduled twice in $\varsigma^2_{\mathcal{O}}$ with a makespan that is either equal to or less than that of $\varsigma^1_{\mathcal{O}}$. □

Proof On one hand, operation o of proposition 2.8 can be scheduled after $C_{\max}(\varsigma^2_S)$. On the other hand, it can be scheduled in the ordered sequence such that is finished before $C_{\max}(\varsigma^2_S)$. Therefore operation o can be scheduled twice consecutively in $\varsigma^2_{\mathcal{O}}$. ■

This effect can be seen in figure 2.7, when operation o_8 is also scheduled at time 5 in figure 2.7b the completion will be scheduled at exactly the same times as they are scheduled in figure 2.7a.

We have seen that all operations of a completion of a dominated solution can be scheduled at the same time, or earlier, in the schedule of a dominating solution. We can also deduce another important property of domination, which considers not the operations individually but the location within the sequence. For this we denote by $\varsigma[i]$ the i-th operation of the solution ς, that is the operation at index i in the sequence of ς. Recall that C_o denotes the finish time of operation o. To prevent any ambiguity we extend this to $C_o(\varsigma)$ to denote the finish time of operation o in solution ς.

Proposition 2.10
Let partial solution ς^1_S of solution $\varsigma^1_{\mathcal{O}}$ be dominated in stage $i = |S|$ by solution ς^2_S. Let $\varsigma^2_{\mathcal{O}}$ be the solution welded from the completion of ς^1_S to $\varsigma^1_{\mathcal{O}}$ and ς^2_S. Then we have that for any $j > i = |S|$ that $C_{\varsigma^2_{\mathcal{O}}[j]}(\varsigma^2_{\mathcal{O}}) \leq C_{\varsigma^1_{\mathcal{O}}[j]}(\varsigma^1_{\mathcal{O}})$. □

Proof When the completion from $\varsigma^1_{\mathcal{O}}$ to ς^1_S is scheduled (possibly idle) at the times of $\varsigma^1_{\mathcal{O}}$ after ς^2_S, the proposition trivially holds. When this schedule is converted to a no-idle schedule, operations are only moved forward. If this conversion is done in unit steps at the time, it is easy to see that the condition holds after each step. When an operation is moved forward by 1 without changing the order of operations, the proposition naturally holds. When two operations must be switched to keep the ordering, they have the same finish time just before the second operation is moved forward. This means that the order of the operations can be changed without changing the finish time at any index. So at each index $j > i$ the finish time can only decrease. ∎

We have seen that domination with a state definition $\xi_{S\wr\vec{\alpha}\wr\vec{\eta}}$ allows for indirect domination. When a partial solution $\varsigma^1_{S\wr\vec{\alpha}^1\wr\vec{\eta}^1}$ is dominated by $\varsigma^2_{S\wr\vec{\alpha}^2\wr\vec{\eta}^2}$ we have the guarantee that for each completion $\varsigma^1_{\mathcal{O}}$ another (welded) solution $\varsigma^2_{\mathcal{O}}$ with equal or lower makespan exists, however, we do not yet have the guarantee that such a solution is found. It is possible that a dominating solution $\varsigma^2_{S\wr\vec{\alpha}^2\wr\vec{\eta}^2}$ did not have an ordered completion with equal or lower makespan as $\varsigma^1_{\mathcal{O}}$. To show that we cannot dominate all optimal solutions we need the following proposition.

Proposition 2.11
Let $\varsigma^1_{\mathcal{O}}$ be a solution and let its partial solution ς^1_S be dominated indirectly by ς^2_S in stage $i = |S|$. Let $\varsigma^2_{\mathcal{O}}$ be the solution welded from ς^2_S and the expansion of ς^1_S to $\varsigma^1_{\mathcal{O}}$. For $k \geq 2$, let $\varsigma^k_{S_k}$ be a partial solution of $\varsigma^k_{\mathcal{O}}$ that is directly or indirectly dominated by a partial solution $\varsigma^{k+1}_{S_k}$. Let $\varsigma^{k+1}_{\mathcal{O}}$ be the solution welded from $\varsigma^{k+1}_{S_k}$ and the expansion from $\varsigma^k_{S_k}$ to $\varsigma^k_{\mathcal{O}}$. When all dominations occur at or before stage i, thus $|S_k| \leq i$, we have $\varsigma^k_{\mathcal{O}} \neq \varsigma^1_{\mathcal{O}}$ for all welded solutions $k \geq 2$. □

Proof Since ς^2_S dominates ς^1_S indirectly there exists an operation $o \in \mathcal{O} \setminus S$ that is scheduled in $\varsigma^2_{\mathcal{O}}$ before the last operation $\Lambda(\varsigma^2_S)$. This operation o is scheduled in $\varsigma^1_{\mathcal{O}}$ such that $\psi(\varsigma^1_{\mathcal{O}}, o) \geq C_{\Lambda(\varsigma^2_S)}(\varsigma^2_S)$. First we conclude that the index of $\Lambda(\varsigma^2_S)$ is at least $i+1$ in $\varsigma^2_{\mathcal{O}}$. Using proposition 2.10 and the fact that all dominations occur before stage $i+1$ we conclude that for all solutions $\varsigma^k_{\mathcal{O}}$ with $k \geq 2$ we have for operation $\varsigma^k_{\mathcal{O}}[i+1]$ that $C_{\varsigma^k_{\mathcal{O}}[i+1]}(\varsigma^k_{\mathcal{O}}) \leq C_{\Lambda(\varsigma^2_S)}(\varsigma^2_{\mathcal{O}})$. When $C_o(\varsigma^k_{\mathcal{O}}) \leq C_{\Lambda(\varsigma^2_S)}(\varsigma^2_{\mathcal{O}})$ we can conclude that $C_o(\varsigma^{k+1}_{\mathcal{O}}) \leq C_{\Lambda(\varsigma^2_S)}(\varsigma^2_{\mathcal{O}})$. When $o \notin S_k$ this follows directly from the domination and when $o \in S_k$ this follows from the fact that we have an ordered sequence, $|S_k| \leq i$ and that $C_{\varsigma^k_{\mathcal{O}}[i+1]}(\varsigma^k_{\mathcal{O}}) \leq C_{\Lambda(\varsigma^2_S)}(\varsigma^2_{\mathcal{O}})$. So in all solutions $\varsigma^k_{\mathcal{O}}$ with $k \geq 2$ we have that operation o finishes before it even starts in $\varsigma^1_{\mathcal{O}}$, and therefore $\varsigma^k_{\mathcal{O}} \neq \varsigma^1_{\mathcal{O}}$. ∎

Such chain of dominated solutions leads to the following corollary.

Corollary 2.12
When partial solution ς^1_S of solution $\varsigma^1_{\mathcal{O}}$ is dominated in the DP algorithm before the last stage in stage $i = |S|$ ($i < |\mathcal{O}|$) there exists a partial solution in stage $i+1$ with a completion with a makespan no larger than that of $\varsigma^1_{\mathcal{O}}$. □

Proof If the completion to $\varsigma^1_{\mathcal{O}}$ of ς^1_S is dominated directly by a partial solution ς^2_S this solution is expanded. So one of its expanded partial solutions in stage $i+1$ must have a completion that has a makespan no larger than that of $\varsigma^1_{\mathcal{O}}$. If ς^2_S dominates ς^1_S indirectly, proposition 2.11 shows that there exists no chain of welded solutions dominated at stages before $i+1$ such that any of the welded solutions is $\varsigma^1_{\mathcal{O}}$. Since there exist a limited number of solutions, only a cycle of domination between such welded solutions can prevent the existence of a partial solution in stage $i+1$. With only direct domination it is clear that no cycle exists, since the domination itself prevents domination in a later stage. With indirect domination we can apply proposition 2.11 at each indirect domination preserving the property of the previous indirect dominations which prevent that one of the previous dominated solutions is found as dominating solution.

Suppose such a cycle of domination between welded solutions with indirect, and possibly direct, domination exists. Let t be the largest stage in which domination occurs in this cycle. Then domination in stage t must by definition be indirect, since otherwise we would have a partial solution in stage $t+1$ in the cycle. Proposition 2.11 directly gives a contradiction to the existence of this cycle. So no such cycle exists and this chain of welded solutions must lead to a partial solution not dominated in at least stage $i+1$. ∎

With these ingredients we can prove the optimality of definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$, which allows for more domination compared to the original state definition $\xi_{S,\vec{\eta} \wr \vec{\alpha}}$.

Proposition 2.13
Using state definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$ leads to an optimal DP algorithm for the JSSP. □

Proof Suppose an optimal solution $\mathring{\varsigma}^1_{\mathcal{O}}$ is dominated, then there is a partial solution $\mathring{\varsigma}^1_S$ of $\mathring{\varsigma}^1_{\mathcal{O}}$ that is dominated in stage $i = |S|$ by another partial solution ς^2_S. If $i < |\mathcal{O}|$ corollary 2.12 provides a partial solution in stage $i+1$ with an optimal completion. Using this iteratively this provides an optimal solution in stage $|\mathcal{O}|$ where it can only be dominated directly by another optimal solution. So the DP algorithm with state definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$ provides an optimal solution. ∎

The algorithm using state definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$ is described in algorithm 2.3. In contrast to algorithm 1.9 the optimal solution can directly be taken from $\hat{\xi}_{\mathcal{O}}$ as it has just a single element. This can be derived from the fact that $\vec{\eta}$ is a zero-dimensional vector for $S = \mathcal{O}$ and the special definition of $\vec{\alpha}$ in this case. The complexity analysis is a bit more complicated for this algorithm. Straightforward calculation of the complexity would give $\mathcal{O}\big(U(U+N)MN2^{MN}\big)$, consisting of: U as an upper bound for the number of non-dominated solutions in any state $\hat{\xi}_S$, MN possible expansions for each solution and 2^{MN} possible subsets of $\mathcal{O}$. Finally, the factor $U+N$ is the effort for each expansion, N for the calculation of the new state variables $\vec{\alpha}$ and $\vec{\eta}$, and U for updating the new state. However, every solution ς_S can only be feasibly expanded to the next operation of each job, thus $o \in \eta(S) \subseteq \varepsilon(S)$ and $|\varepsilon(S)| \leq N$ losing a factor M.

Algorithm 2.3 Forward DP algorithm for the JSSP

Input: An instance of the JSSP defined by a set of operations $\mathcal{O}$, with for each operation a machine $m(o)$ and a processing time $p(o)$

Output: The ordered sequence corresponding to an optimal solution

$\hat{\xi}_\emptyset = \left\{ \varsigma_{\emptyset \wr \vec{0} \wr \vec{0}} = \langle\ \rangle \right\}$

for $L = 0$ **to** $|\mathcal{O}| - 1$ **do**
    for all $S \subset \mathcal{O}$ **such that** $|S| = L$ **do**
        for all $\varsigma_{S \wr \vec{\alpha} \wr \vec{\eta}} \in \hat{\xi}_S$ **do**
            for all $o \in \varepsilon(S)$ **do**
                if $\eta(\varsigma_{S \wr \vec{\alpha} \wr \vec{\eta}}, o) = 1$ **then** // Feasibility: expansion is ordered
                    $\varsigma = \varsigma_{S \wr \vec{\alpha} \wr \vec{\eta}} \diamond\!\!\Longrightarrow o$
                    if $\varsigma \not\mathrel{\dot{\leq}} \varsigma', \forall \varsigma_i \in \hat{\xi}_{S \cup \{i\}}$ **then**
                        $D = \left\{ \varsigma' \in \hat{\xi}_{S \cup \{i\}} \;\middle|\; \varsigma \mathrel{\dot{\geq}} \varsigma' \right\}$ // All solutions dominated by ς
                        $\hat{\xi}_{S \cup \{i\}} = \{\varsigma\} \cup \hat{\xi}_{S \cup \{i\}} \setminus D$

$\mathring{\varsigma} \in \hat{\xi}_{\mathcal{O}}$

return $\mathring{\varsigma}$

Furthermore, we have N precedence relations of length M for the order of the operations in each job, this removes a factor $\left(\frac{2^M}{M+1}\right)^N$ in the possible subsets of $\mathcal{O}$ for which the state has feasible solutions, see section 4.2.1. These reductions lead to a complexity of $\mathcal{O}\big(U(U+N)N(M+1)^N\big)$.

To estimate $U = \max_{S \subset \mathcal{O}} |\hat{\xi}_S|$ we first conclude that the values for $C_{\max}(\varsigma_{S \wr \vec{\alpha}})$ are limited for any solution $\varsigma_{S \wr \vec{\alpha}} \in \hat{\xi}_S$. The values of $\vec{\alpha}$ for any solution ς are by definition in the interval $[C_{\max}(\varsigma), C_{\max}(\varsigma) + p_{\max}]$, where $p_{\max} = \max_{o \in \mathcal{O}} p(o)$. Let now $\varsigma_{S \wr \vec{\alpha}}$ be such that $C_{\max}(\varsigma_{S \wr \vec{\alpha}}) = \min_{\varsigma \in \hat{\xi}_S} C_{\max}(\varsigma)$, since $\alpha(\varsigma_{S \wr \vec{\alpha}}, o) \leq C_{\max}(\varsigma_{S \wr \vec{\alpha}}) + p_{\max}$ we conclude that any solution $\varsigma'_{S \wr \vec{\alpha}}$ with $C_{\max}(\varsigma'_{S \wr \vec{\alpha}}) \geq C_{\max}(\varsigma_{S \wr \vec{\alpha}}) + p_{\max}$ is dominated by $\varsigma_{S \wr \vec{\alpha}}$ and thus $\varsigma'_{S \wr \vec{\alpha}} \notin \hat{\xi}_S$. So $C_{\max}(\varsigma)$ for $\varsigma \in \hat{\xi}_S$ can have at most $p_{\max}$ different values. By definition two solutions in $\varsigma_{S \wr \vec{\alpha}}, \varsigma'_{S \wr \vec{\alpha}'} \in \hat{\xi}_S$ do not have equal aptitude values ($\vec{\alpha} \neq \vec{\alpha}'$). So for each value of $C_{\max}$ we have at most $(1 + p_{\max})^N$ different solutions, this leads to the following estimate: $U \leq p_{\max}(1 + p_{\max})^N$.

To improve this bound we look at the maximum number of possible values of $\vec{\alpha}$ for solutions with the same $C_{\max}$ that do not allow for any domination. For all solutions $\varsigma \in \hat{\xi}_S$ with $C_{\max}(\varsigma) = c$ conclude that $0 \leq \alpha(\varsigma, o) - c \leq p_{\max}$ for all $o \in \varepsilon(S)$ (all values of $\vec{\alpha}$). Each aptitude vector $\vec{\alpha}$ can be represented by a subset of the multiset $\mathcal{S} = k_1, \ldots, k_N$, with $k_i = p_{\max}$ for $i = 1, \ldots, N$. $\mathcal{S}$ consists of $k_i = p_{\max}$ copies of N different elements x_i, $i = 1, \ldots, N$, where x_i is associated with job i. Denote by $\sigma(\varsigma)$ the subset associated with the aptitude of ς. Thus, $\sigma(\varsigma) \subseteq \mathcal{S}$ is composed by taking $\forall o \in \varepsilon(S)$, $\alpha(\varsigma, o) - c$ copies of x_i where $i = j(o)$. Now observe that for $\varsigma, \varsigma' \in \hat{\xi}_S$, with $C_{\max}(\varsigma) = c = C_{\max}(\varsigma')$, $\varsigma \leq \varsigma'$ if and only if $\sigma(\varsigma) \subseteq \sigma(\varsigma')$ with $\sigma(\varsigma) = \sigma(\varsigma')$ when $\varsigma \doteq \varsigma'$. We conclude that $\forall \varsigma, \varsigma' \in \hat{\xi}_S$, with $C_{\max}(\varsigma) = c = C_{\max}(\varsigma')$, we have $\sigma(\varsigma) \nsubseteq \sigma(\varsigma')$ and $\sigma(\varsigma) \nsupseteq \sigma(\varsigma')$.

Intermezzo: Antichains

Before proceeding with our analysis, we recall the concept of an antichain, see Anderson [3, chaps. 4.3 & 9.4] for further details and proofs.

Definition 2.14
An Antichain is a collection of subsets of a set where no two elements of the collection are subsets of each other. □

Proposition 2.15
The largest antichain in the collection of all subsets of a multiset is smaller or equal to the largest rank number N_i (N_i the number of elements with rank, or size, i). □

Proposition 2.16
The size of the largest rank number for a multiset is equal to size of the middle-rank number λ which is

$$N_\lambda \approx \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \frac{\prod_i (k_i + 1)}{\sqrt{\frac{1}{3}\sum_i k_i(k_i + 2)}}.$$

□

According to the definition of an antichain the sets $\sigma(\varsigma)$ for $\varsigma \in \hat{\xi}_S$ and $C_{\max}(\varsigma) = c$ form an antichain of $\mathcal{S}$. Since we have $k_i = p_{\max}$ for all i, the maximum size of such antichain is approximately $\left(\frac{2}{\pi}\right)^{\frac{1}{2}} \frac{(p_{\max}+1)^N}{\sqrt{\frac{1}{3}N({p_{\max}}^2+2p_{\max})}}$. As we have $p_{\max}$ of these antichains, one for every possible value of $C_{\max}(\varsigma)$, we conclude that

$$U \leq p_{\max} \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \frac{(p_{\max} + 1)^N}{\sqrt{\frac{1}{3}N({p_{\max}}^2 + 2p_{\max})}} = \mathcal{O}\left(\frac{{p_{\max}}^{N+1}}{\sqrt{N{p_{\max}}^2}}\right) = \mathcal{O}\left(\frac{{p_{\max}}^{N}}{\sqrt{N}}\right).$$

From this we can obtain an upper bound on the total time complexity of the

DP algorithm for the JSSP

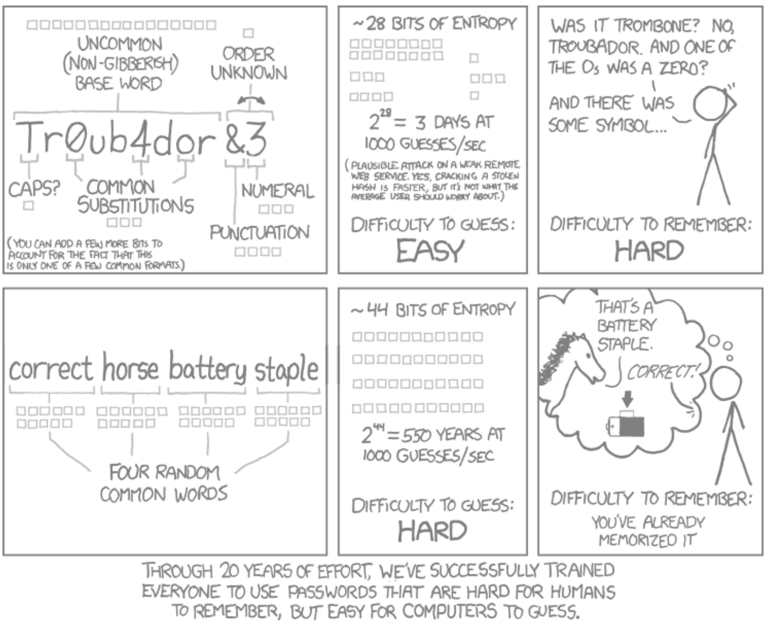
$$
\begin{aligned}
&\mathcal{O}\left(\frac{{p_{\max}}^N}{\sqrt{N}}\left(\frac{{p_{\max}}^N}{\sqrt{N}}+N\right)N(M+1)^N\right)\\
&\mathcal{O}\left(\left(\frac{{p_{\max}}^{2N}}{N}+\frac{N{p_{\max}}^N}{\sqrt{N}}\right)N(M+1)^N\right)\\
&\mathcal{O}\left(\left({p_{\max}}^{2N}+N\sqrt{N}{p_{\max}}^N\right)(M+1)^N\right)\\
&\mathcal{O}\left({p_{\max}}^{2N}(M+1)^N\right).
\end{aligned}
$$

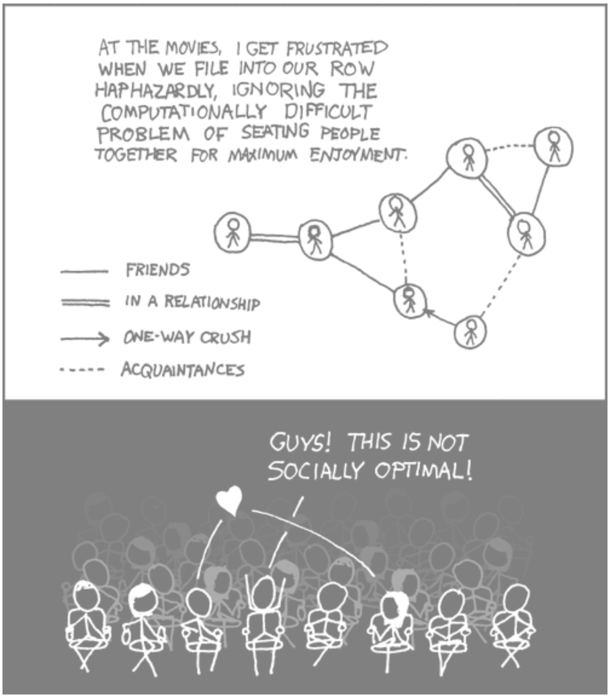
Although the upper bound $U = \mathcal{O}\left(\frac{{p_{\max}}^N}{\sqrt{N}}\right)$ gives a complexity of

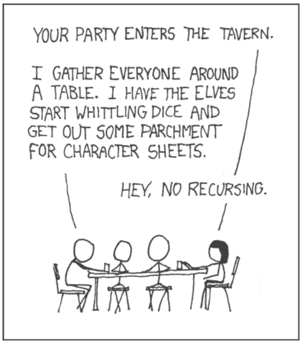
$$\mathcal{O}\left({p_{\max}}^{2N}(M+1)^N\right),$$

experimental results suggest that the actual value is just a small part of this bound, see section 5.2.1

2




AT THE MOVIES, I GET FRUSTRATED WHEN WE FILE INTO OUR ROW HAPHAZARDLY, IGNORING THE COMPUTATIONALLY DIFFICULT PROBLEM OF SEATING PEOPLE TOGETHER FOR MAXIMUM ENJOYMENT.
FRIENDS
IN A RELATIONSHIP
ONE-WAY CRUSH
ACQUAINTANCES
GUYS! THIS IS NOT SOCIALLY OPTIMAL!



THREE

The Dynamic Programming State Space

Before we continue with the classical problems introduced in the previous chapter and their variants in the next chapters,we take a better look at the DP state space in general. To improve the performance we can add bounding such that the size of the state space of a DP algorithm can be reduced while preserving optimality. This works in a similar way as in Branch and Bound as was first done by Marsten and Morin [82]. In section 3.2 we show how to alter the state space definition in such a way that known optimal solutions can be disregarded enabling new optimal solutions to be found. We show that using this strategy iteratively, including newly found solutions in each iteration, eventually will produce all optimal solutions. Finally, we investigate possibilities to modify the optimal DP algorithm into a heuristic such that practical running times can be achieved.

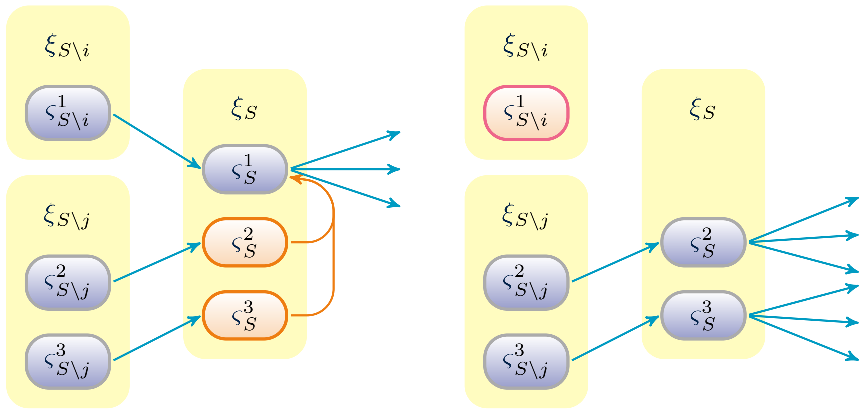
3.1 Dynamic bounding

To improve the performance of a DP algorithm we can add bounding to each state of the DP algorithm. This was first suggested for the TSP and other sequencing problems by Marsten and Morin [82] and later used by Carraway and Schmidt [27], Dyer, Riha, and Walker [41] and Puchinger and Stuckey [99] for other combinatorial problems. This can be done very similarly to the well-known principle of Branch and Bound. However, the effects of bounding on a DP state space can be very different from the effects on a Branch and Bound algorithm.

As in Branch and Bound, we try to find an upper bound $\mathcal{UB}$ on the problem's value and construct a lower bound $\mathcal{LB}(\varsigma_S)$ for any completion of each partial solution ς_S. Naturally, this lower bound is also a lower bound for all possible expansions and completions of ς_S. So if $\mathcal{LB}(\varsigma_S) > \mathcal{UB}$ we can prune the state space of DP by discarding the partial solution ς_S as soon as it is created.

This saves the possibly exponential effort of considering all expansions of ς_S. Nevertheless, such bounding in a DP state space can have negative running time performance influences as well, since such bounds need to be calculated. This is analogous to Branch and Bound. However, in contrast to a Branch and Bound tree where a pruned node has no effect on the other nodes evolved from an earlier branch, in a DP state space removing a node can have effect on other parts of the state space as well.

To illustrate this we show an example in figure 3.1. Consider two solutions



a: ς^1_S *dominates* ς^2_S *and* ς^3_S **b:** ς^1_S *is not created*

Figure 3.1: *Possible negative effect of bounding*

$\varsigma^2_{S\setminus j}$ and $\varsigma^3_{S\setminus j}$ in state $\xi_{S\setminus j}$ that do not dominate each other (Naturally, since we have multiple solutions in a state, we have some γ for these states and solutions, which is disregarded as it is besides this observation irrelevant). Consider their expansions, with j, to ς^2_S and ς^3_S, respectively, in state ξ_S, where these solutions still do not dominate each other. Assume that ς^2_S and ς^3_S are dominated in ξ_S by ς^1_S, which is an expansion of $\varsigma^1_{S\setminus i}$ in $\xi_{S\setminus i}$ with i, after which only expansions of ς^1_S are considered. When $\varsigma^1_{S\setminus i}$ is bounded, $\mathcal{LB}(\varsigma^1_{S\setminus i}) > \mathcal{UB}$, the expansion ς^1_S is not created and ς^2_S as well as ς^3_S are not dominated. Now all the expansions of both ς^2_S and ς^3_S are considered, which is potentially much more effort. The difference in respect to Branch and Bound is that the DP state space forms a directed acyclic graph in contrast to the tree formed by Branch and Bound.

Naturally, the expansions of ς^2_S and ς^3_S should not be considered, since $\mathcal{LB}(\varsigma^1_S) \geq \mathcal{LB}(\varsigma^1_{S\setminus i}) > \mathcal{UB}$ and $\varsigma^1_S \geq \varsigma^2_S, \varsigma^3_S$, there should be lower bounds $\mathcal{LB}(\varsigma^2_S)$, $\mathcal{LB}(\varsigma^3_S)$ such that $\mathcal{LB}(\varsigma^1_S) \leq \mathcal{LB}(\varsigma^2_S), \mathcal{LB}(\varsigma^3_S)$. To prevent this possible negative effect, this principle gives us the only requirement for the lower bound — except that it is indeed a lower bound for all completions — when $\varsigma^1 \geq \varsigma^2$ we should find bounds such that $\mathcal{LB}(\varsigma^1) \leq \mathcal{LB}(\varsigma^2)$. In the previous example we would have that ς^2_S would be bounded because $\mathcal{LB}(\varsigma^2_S) \geq \mathcal{LB}(\varsigma^1_S) \geq \mathcal{LB}(\varsigma^1_{S\setminus i}) > \mathcal{UB}$, similarly ς^3_S would be bounded.

The simplest way to assure this property for a lower bound is to create a lower bound that only depends on the state variables, ϕ and γ, such that variables in γ in the direction of increasing dominance have a decreasing effect on the lower bound. So when $\geq = \{\leq, \leq\}$, the two variables in γ dominate for lower values so decreasing these values should have a decreasing effect on the lower bound. Of course, not all variables of ϕ and γ have to be used in the calculation of a a lower bound. Furthermore, the dependance of the lower bound on the state variables has only to be theoretical, performance can in some occasions greatly benefit from extra intermediate variables for solutions.

Notice that such a lower bound for solutions is not necessarily a bound on states. If a lower bound depends on variables in γ, different solutions in the same state can have different lower bounds.

The size of the DP state space can be reduced with bounding which can improve the performance of a DP algorithm as it is likely that less states have to be evaluated. However, the total performance will depend on the performance of calculation of a lower bound and on possible effects as explained above. As section 5.2.3 shows, dynamic bounding can be very effective.

3.2 Finding all optimal solutions with DP

Some problems have multiple optimal solutions, for example any symmetric TSP (with $n > 2$) has at least two optimal solutions as any optimal solution can also be traveled in the opposite direction. Also JSSP instances have typically multiple solutions as operations that are not on the critical path can often be swapped. A DP algorithm typically finds only a single solution. When two (partial) solutions are equal only one is regarded as not dominated, so even if there are multiple equal solutions in the last stage (symmetric TSP) only one is found. However, just keeping equal partial solutions will not always be sufficient to find all optimal solutions. Within the state space for the JSSP full domination can occur between two partial solutions which both have an optimal completion. Also effort can be wasted when equal solutions, that have no optimal completion, are both kept.

Proposition 3.1
To find optimal solutions other than the ones already known we can run the original DP algorithm while preventing domination by partial solutions for which optimal completions are known. □

Proof We first observe that for any optimal solution $\mathring{\varsigma}$ not found by the original DP algorithm some partial solution ς that can be completed to $\mathring{\varsigma}$ must be dominated by some other partial solution ς' in some stage k. From the very nature that an optimal solution is dominated we can conclude that the partial solution ς' dominating ς must have at least one optimal completion $\mathring{\varsigma}'$. If $\mathring{\varsigma}'$ is also not found by the DP algorithm it must be dominated by an other partial solution ς'' in some stage l with $l > k$, which in turn must have an optimal completion $\mathring{\varsigma}''$. Following these optimal solutions with dominated partial solutions will finally lead to the optimal solution found by the DP algorithm. A circular relation of dominating optimal solutions is impossible, since these dominations occur in strictly increasing stages of the original DP state space. Similar to the reasoning in the proof of corollary 2.12 domination must occur in different stages of the state space. For example, if a circular domination between three solutions occurs in stages $a < b < c$, the domination in stage b would not be possible, since the dominating solution would itself already be dominated in stage a.

Since none of the not yet found optimal solutions can be dominated by partial solutions of already found optimal solutions, at least one of the not yet found optimal solutions has to be found. ∎

3

To incorporate this in an original DP algorithm we first number all found optimal solutions with a unique identifier, for simplicity we assume the optimal solutions are simply numbered increasingly by the order they are found. To the original DP algorithm we add a new variable Ω to ϕ in the state definition leading to a state definition of $\xi_{S,\Omega \wr \gamma}$ (originally $\phi = \{S\}$). We define Ω as a set of identifiers of optimal solutions which can be completed with the current partial solution. The algorithm starts with an empty set S and the identifiers of all optimal solutions in Ω.

The recurrence relation depends of course on the recurrence relation of the original DP. The following relation expresses the recurrence relation regarding Ω, a solution state with a non empty Ω can only be an extension of a partial solution from the same optimal solution(s). This solution is only feasible if the current extension is indeed the extension leading to the optimal solutions in Ω.

$$C\left(\check{\xi}_{S,\Omega}\right) = \begin{cases} \min\limits_{i \in S} \left\{C\left(\check{\xi}_{S\setminus\{i\},\Omega'} \Diamond\!\!\Rightarrow i\right)\right\} & \text{if } \Omega = \emptyset \\ C\left(\check{\xi}_{S\setminus\{i\},\Omega'} \Diamond\!\!\Rightarrow i\right) & \text{otherwise, where } \Omega \subseteq \Omega' \text{ and } i \text{ the} \\ & \text{correct extension for the solutions in } \Omega \end{cases}$$

Figure 3.2 shows the effect of Ω on the state space. In figure 3.2a we have no optimal solutions and in figure 3.2b the dominating solution is optimal. Adding $\Omega$to the state definition isolates solution “1” in its own state.

Proposition 3.2
Adding Ω to the state definitions maintains the optimality principle and prevents any domination by partial solutions of known optimal solutions □

Proof Since all found optimal sequences are known before we perform the DP algorithm, we can find for each expansion $\varsigma_{S,\Omega \wr \gamma} \Diamond\!\!\Rightarrow i$ the new set Ω' by taking identifers of optimal solutions that are identified by Ω and where i is at position $|S| + 1$ in the optimal sequence. This addition to the state definition maintains the optimality principle, since the new values of Ω for an expansion can be derived from the original state variables and the choice of the expansion.

Also the addition of Ω prevents any domination by partial solutions of optimal solutions as they are singled out by this state definition. An identifier

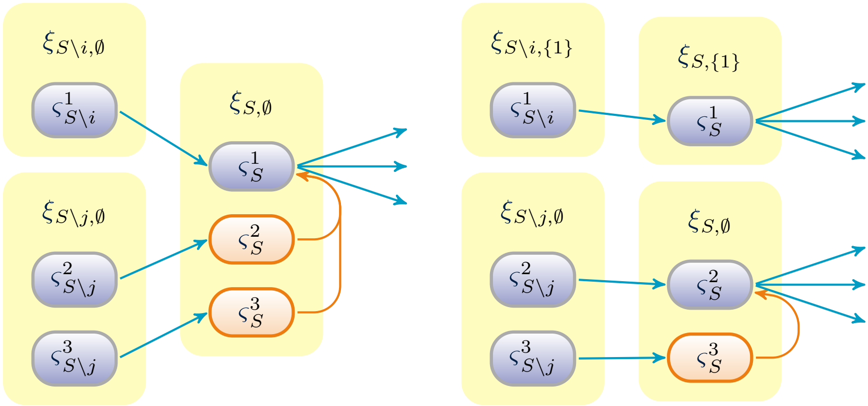


Figure 3.2: *The effect of* Ω *in the state space*

of an optimal solution will only occur in Ω of a single solution in each stage, preventing any domination by this partial solution. We conclude that this state definition allows only domination between partial solution for which no optimal completion is known and $\Omega = \emptyset$. ∎

Now all optimal solutions can be found by running the DP algorithm with Ω in the state definition iteratively until no new optimal solutions are found.

To show the effect of Ω on a total state space we use a small example of the linear assignment problem. In figure 3.3 the DP state space for the linear assignment problem specified in table 3.1 is given. The found optimal solution we call A and figure 3.4 gives the new DP state space resulting in the second optimal solution.

A sketch of the algorithm for finding all optimal solutions is given in algorithm 3.1, where $\text{DP}_\Omega(\Sigma)$ is an original DP algorithm altered by adding Ω to ϕ in the state definition. It takes a set of optimal solutions Σ to be assigned identifiers and used in Ω. It returns a set of found optimal solutions. To speed up this algorithm the original DP algorithm can be altered slightly to find multiple optimal solutions in a single run. This can be done by altering the definition of $\hat{\xi}$ slightly allowing for multiple solutions with $\varsigma \doteq \varsigma'$. When there was a single

	t_1	t_2	t_3	t_4
e_1	5	7	2	11
e_2	3	2	11	1
e_3	3	5	9	9
e_4	9	11	3	9

Table 3.1: *Instance of the linear assignment problem*

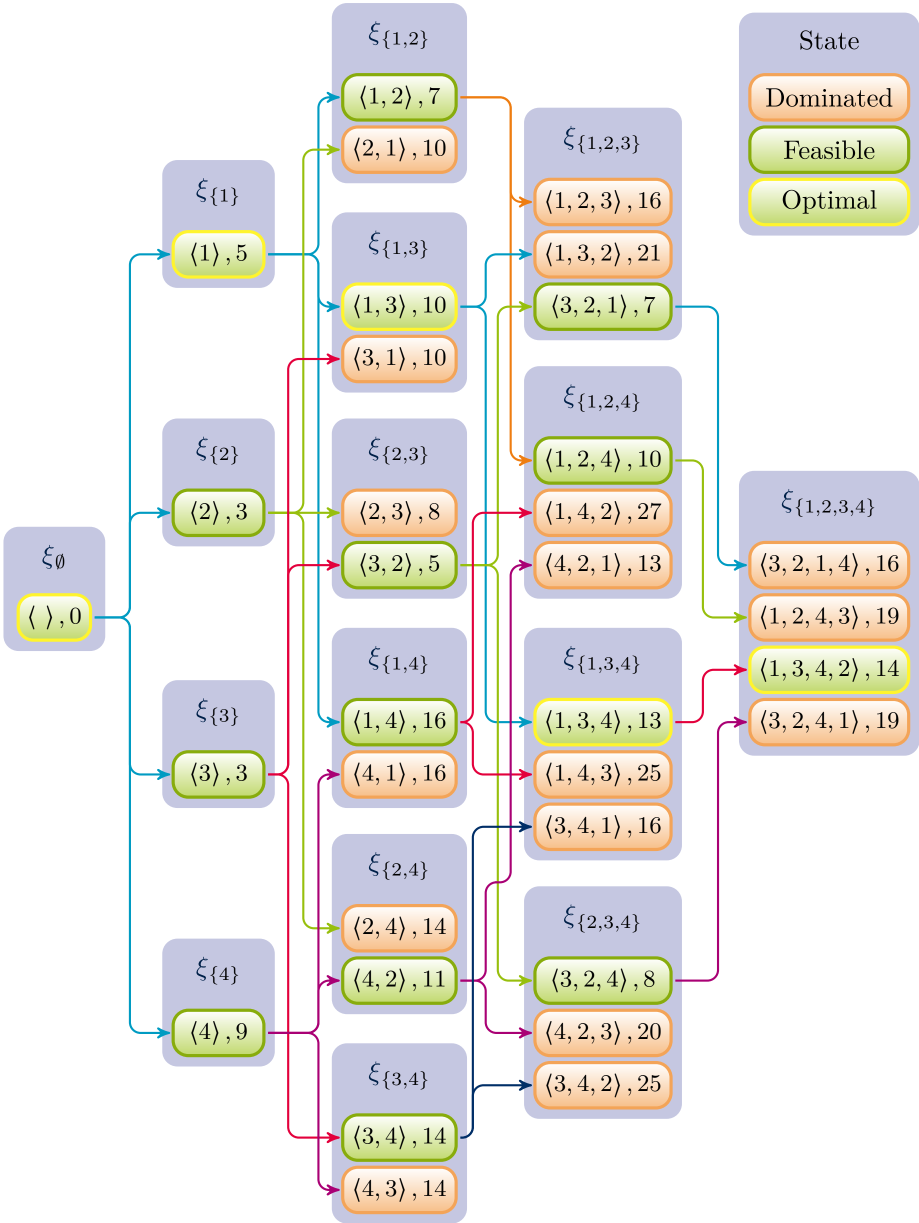


Figure 3.3: *State space of DP for the linear assignment problem in table 3.1*

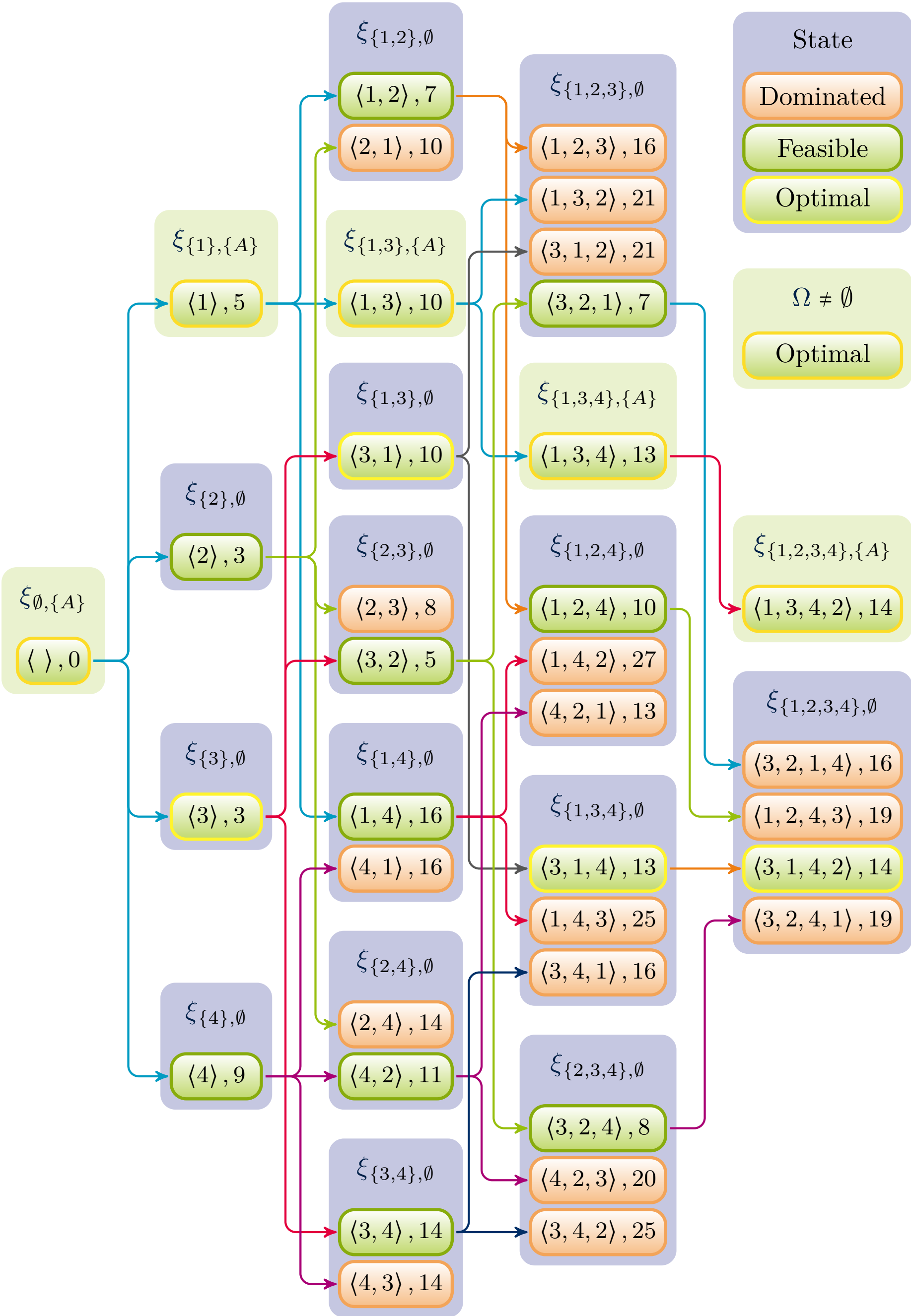


Figure 3.4: *State space of DP for the linear assignment problem in table 3.1 with $\langle 1,3,4,2\rangle$ as optimal solution A*

Algorithm 3.1 Iterative algorithm to find all optimal solutions using DP

Input: Equal to the original DP algorithm
Output: All optimal solutions of the original problem

$\Sigma = \emptyset$

repeat
  $\Sigma' = \mathrm{DP}_\Omega(\Sigma)$
  $\Sigma' = \Sigma' \setminus \Sigma$
  $\Sigma = \Sigma \cup \Sigma'$
until $\Sigma' = \emptyset$

return Σ

optimal solution per state in the original DP algorithm ($\check{\xi}$), multiple solutions with the same minimum (or maximum) are allowed.

3

The complexity of this algorithm is mostly determined by the complexity of the original DP algorithm and the number of times this algorithm is performed. This depends on the number of new solutions found in each iteration of $DP_\Omega(\Sigma)$, but is bounded by the number of found optimal solutions $f = |\Sigma|$. The alterations to the original DP algorithm normally have no effect on the complexity. The number of states in the state space is maximally increased by nf, where f is the number of optimal solutions and $n = |\mathcal{S}|$ is the number of nodes the original DP algorithm is performed on. This is easy to see as each known optimal solution has a single state for each stage in the DP state space, multiple optimal solutions having the same partial solution in the beginning share such states. Note that all Ω of states of a single stage are disjoint and that their union is equal to all known optimal solutions. As we now have $nf + 2^n$ instead of 2^n states we could write $\mathcal{O}(DP_\Omega) = \mathcal{O}(DP) + \frac{nf}{2^n}\mathcal{O}(DP)$. As long as the number of optimal solutions is not exponential, this is largely dominated by the exponential number of states resulting from the subsets of $\mathcal{S}$.

Furthermore, the number of solutions per state is increased by keeping multiple solutions with $\varsigma \doteq \varsigma'$. As we take a bound of f for the number of times the original DP algorithm has to be performed, we disregard this speedup also in the complexity of the original DP algorithm. Only having many solutions with equal dominance values would dramatically decrease the running time of the original DP algorithm, in this case we generally expect also a lot of optimal solutions. In the worst-case scenario — where all partial solutions per state are equal, and thus all solutions are optimal — adding this speed-up would result in a brute-force algorithm enumerating all solutions. However, not using this speedup would result in performing the original DP algorithm once for each of these solutions. All in all the effort of finding all optimal solutions is at most the effort of the original DP algorithm for each found solution, thus $f\mathcal{O}(\mathrm{DP})$.

If there is sufficient memory, the performance can be improved by keeping the complete state space in memory and after each iteration the partial solutions of the newly found optimal solutions could be removed from their corresponding states. For partial solutions in a state with $\Omega \neq \emptyset$ this would result in replacing the original state containing a single solution with a new state where the new identifier is added to Ω. For partial solutions in states with $\Omega = \emptyset$ this would result in the removal of the partial solution from the state, which is placed into its own state where Ω contains the identifier of the new optimal solution. The domination for the remaining state $\check{\xi}$, or $\hat{\xi}$ depending on the original DP algorithm, should be reevaluated. The DP algorithm can now be performed by only expanding $\check{\xi}$, or new non-dominated solutions in $\hat{\xi}$, and their expansions. Note that, these expansions can belong to already existing states, in this case $\check{\xi}$, or $\hat{\xi}$, have to be reevaluated and any new solution $\check{\xi}$, or in $\hat{\xi}$, should be expanded. This prevents the reevaluation of the same expansions in each iteration of the original DP algorithm. However, to accommodate for the reevaluation of $\hat{\xi}$, or $\check{\xi}$, all solutions in ξ have to be kept in memory. Once dominated solutions can become non-dominated when the solution dominating it turns out to have an optimal completion.

Note that for the second DP state definition in section 2.3 the addition of Ω to the state definition is not sufficient to be able to find all optimal solutions. The addition of the bookkeeping variables β allows for indirect domination. With indirect domination we allow a partial solution of an optimal solution to be dominated by a partial solution possibly without an optimal completion, if it can be guaranteed that another partial solution with optimal completion would dominate this partial solution. Finding all optimal solutions with this state definition is described in section 5.3.

3.3 Heuristic DP algorithms

The running time of an optimal DP algorithm over sets is often impractical as it is often exponential. To reduce the running time the optimal DP algorithm can be converted into a heuristic algorithm. This section describes basic ways to do this which can be used simultaneously.

3.3.1 Removing state variables

To reduce the size of the state space, variables can be left out of the state definition. For example for the Traveling Salesman Problem with Time Windows (TSPTW) when minimizing on distance d we have an optimal state definition of $\xi_{S,l \wr d,t}$, see section 4.3.6 for a description on time-windows.

To reduce the number of solutions expanded each state we can remove the time from the state definition and change the state definition to $\xi_{S,l \wr d}$. The number of states is not changed but the number of non-dominated solutions per state is reduced, in this case to at most one.

This technique has similarities with state space relaxation using surrogates of state variables as in Christofides, Mingozzi, and Toth [33]. However, there are also some fundamental differences. State space relaxation yields lower bounds since feasibility is not pursued and optimality is enforced in the relaxed problem. Our removal of state variables leads to a heuristic framework producing upper bounds.

We can directly see that the state definition $\xi_{S,l \wr d}$ is not optimal as the feasibility cannot be checked on state variables. Since the DP algorithm minimizes the distance, with state definition $\xi_{S,l \wr d}$ the waiting time in solutions is ignored. This makes it possible that the optimal solution is not found. It is even possible that no feasible solution is found when feasible solutions exist.

To illustrate this we take a look at a small example with five nodes, where node 1 is the depot. The time traveled is taken equal to the distance.

	Distance					Time Window	Service Time
	1	2	3	4	5		
1	-	1	5	20	20	0-100	0
2	20	-	1	5	20	50- 60	5
3	20	5	-	1	20	40- 70	5
4	20	20	20	-	1	50-100	5
5	1	20	20	20	-	60- 71	5

Table 3.2: *Small instance of the TSP with Time Windows*

Obviously, solution $\langle 2,3,4,5,1 \rangle$, with distance 5, is optimal in distance. However, it is not feasible as the visit at node 5 would occur at time $(68,73)$. Solution $\langle 3,2,4,5,1 \rangle$ with distance 17 is feasible, with a visit at time $(66,71)$ at node 5. The state definition $\xi_{S,l \wr d}$ would miss the optimal solution as at state $\xi_{\{2,3,4\},4}$ solution $\langle 2,3,4 \rangle$ would dominate solution $\langle 3,2,4 \rangle$.

A better option to create a heuristic would be using state definition $\xi_{S,l \wr t}$ and minimize on the time. Since distance and time have a large correlation — for traveling they are even equal for this instance — minimizing on time would also minimize the distance as a side effect. Minimizing on time does take the waiting time into account and at state $\xi_{\{2,3,4\},4}$, solution $\langle 3,2,4 \rangle$ would now dominate solution $\langle 2,3,4 \rangle$. Naturally this is still a heuristic, and setting the time window at node 5 to $[60,75]$ would make solution $\langle 2,3,4,5,1 \rangle$ feasible, but it would not be found. Even setting the distance from node 1 to node 3 to 41 would still result in finding solution $\langle 3,2,4,5,1 \rangle$, now with distance 58.

Often multiple state variables have a large correlation, removing some of them will result in a heuristic which when done carefully can still find good solutions.

3.3.2 Limiting the number of expansions

Another way of limiting the number of evaluated partial solutions is limiting the number of expansions for each non-dominated solution. For each non-dominated solution we limit the number of expansions by stopping after finding E feasible expansions. To find the most promising expansions, all possible nodes to expand need to be sorted before performing the actual expansions. This limitation is similar to beam search (Bisiani [19]). However, beam search is applied to the *solution space*, whereas we use a limitation on the search through the *state space*.

For the TSP and VRP the possible expansions could be sorted in increasing order of the distances from the last node in the sequence. For the TSP and VRP this limitation of the number of expansions is reasonable, because edges in the optimal solution will most likely be between two nodes that are near neighbors of each other, as observed by Rego and Glover [102] and Toth and Vigo [115]. For the JSSP the operations to be expanded could be sorted in decreasing order of their tail. However, for the JSSP the number of feasible expansions is already limited to N of the NM possible operations, since per job at most one expansion will be feasible.

This limitation basically reduces a single factor in the complexity to a constant E. For example, the factors n, $n+m$ and N are reduced for the TSP, VRP and JSSP, respectively. The effect of this limitation can be that some states will not contain any feasible solution. Since the number of feasible partial solutions in the following state is at most multiplied by E, the number of feasible partial solutions evaluated will be maximized at E^s for stage s. The total number of feasible partial solutions evaluated will be at most $2E^{|\mathcal{S}|}$ for any $E > 1$. The extreme case $E = 1$ will result in a nearest-neighbor for the TSP. The sorting of possible expansions can often be done as preprocessing, for example for the TSP per node all other nodes can be sorted by distance, otherwise a factor $\log(E)$ will be added to the complexity.

Another way to limit the number of expansions is to use an a-priori measure to include or exclude an expansion. For example, for the TSP and VRP the distance between two nodes can be used. This would mean we would only expand to nodes which are at most a certain distance from the last node. Naturally, a combination of these two, where a minimal number of expansions is also set, would also be possible. A downside of this strategy is that the complexity of this algorithm is possibly equal to the complexity original DP algorithm.

3.3.3 Limiting the number of solutions to expand

Similar to limiting the number of expansions for each solution, the number of solutions to be expanded can be limited, as proposed by Malandraki and Dial [81]. If we limit the number of solutions expanded from each stage to H the computational complexity of the algorithm can be bounded by a polynomial. We call H the width of the stage space. Each stage at most H will be expanded to at most all $n = |\mathcal{S}|$ nodes resulting in at most nH new solutions of which at most H will be expanded. Naturally, we need to decide which solutions to expand, for this we choose the H most promising solutions. How the most promising solutions are determined depends greatly on the characteristics of the problem, and the variables already calculated for each solution. However, the solutions always have to be sorted according to these criteria adding, when using tree sort, a factor $\log(H)$ to the computational complexity. When using the current cost of a solution as sorting criterion for H is used DP naturally tends to get too greedy in the first part of the solutions. For the TSP, using this criterion, setting $H = 1$ will again result in a nearest-neighbor. When using dynamic bounding (section 3.1) the current bound of a solution is a more stable criterion as it also considers nodes not yet in the partial solution. In sections 4.1, 5.2 and 6.6 we will go into further detail of selecting the most promising solutions.

Setting H to a higher value can decrease the solution quality. As more partial solutions are found the partial solution of the current best solution can be left unexpanded. To illustrate this, we use the instance of a linear assignment problem given in table 3.3. The current cost of a partial solution will be used

	t_1	t_2	t_3	t_4
e_1	2	7	2	11
e_2	3	2	11	1
e_3	3	5	9	9
e_4	9	11	3	9

Table 3.3: *Instance of the linear assignment problem*

as criterion to bound the partial solutions of a single stage. The total DP state space for this instance is given in figure 3.5. This is all similar to the example used in section 1.2.1, with different costs. When H is set to $H = 1$ only solutions $\underline{\langle\ \rangle}$, $\underline{\langle 1\rangle}$, $\langle 2\rangle$, $\langle 3\rangle$, $\langle 4\rangle$, $\underline{\langle 1,2\rangle}$, $\langle 1,3\rangle$, $\langle 1,4\rangle$, $\langle 1,2,3\rangle$, $\underline{\langle 1,2,4\rangle}$, $\langle 1,2,4,3\rangle$ are created of which only the underlined solutions are expanded. The final solution $\langle 1,2,4,3\rangle$ has cost 16. When H is set to $H = 2$ the optimal solution is found. Figure 3.6 depicts the state space of DP with $H = 2$, all partial solutions of the original state space are shown. The purple solutions are abandoned, not expanded, due to the limitation to the width H, the grey solutions are not created in this state space.

When H is set to the higher value $H = 3$ again solution $\langle 1,2,4,3\rangle$ is found, as can be seen in figure 3.7. The partial solution $\langle 1,3,4\rangle$ is abandoned since partial solutions $\langle 3,2,1\rangle$, $\langle 1,2,4\rangle$ and $\langle 3,2,4\rangle$ all have a lower cost.

Although a higher value for width H results in evaluating more partial solutions, as long as the stage width is still limitative, it is possible that this abandons the best solution found by a more limitative stage width. The same principle applies to bound E, which bounds the number of expansions per partial solution, as described in the previous section.

3.3.4 Heuristic bounding

When a bounding described in section 3.1 is applied, the used bound for a partial solution must be a lower bound on all possible completions of this partial solution. The calculation of such a bound, sufficient enough to limit the size of the state space, can be difficult or too expensive to achieve a good performance. Possibly, a quick estimation can be used to achieve an estimated value for the optimal completion instead of a valid lower bound. When this value is used

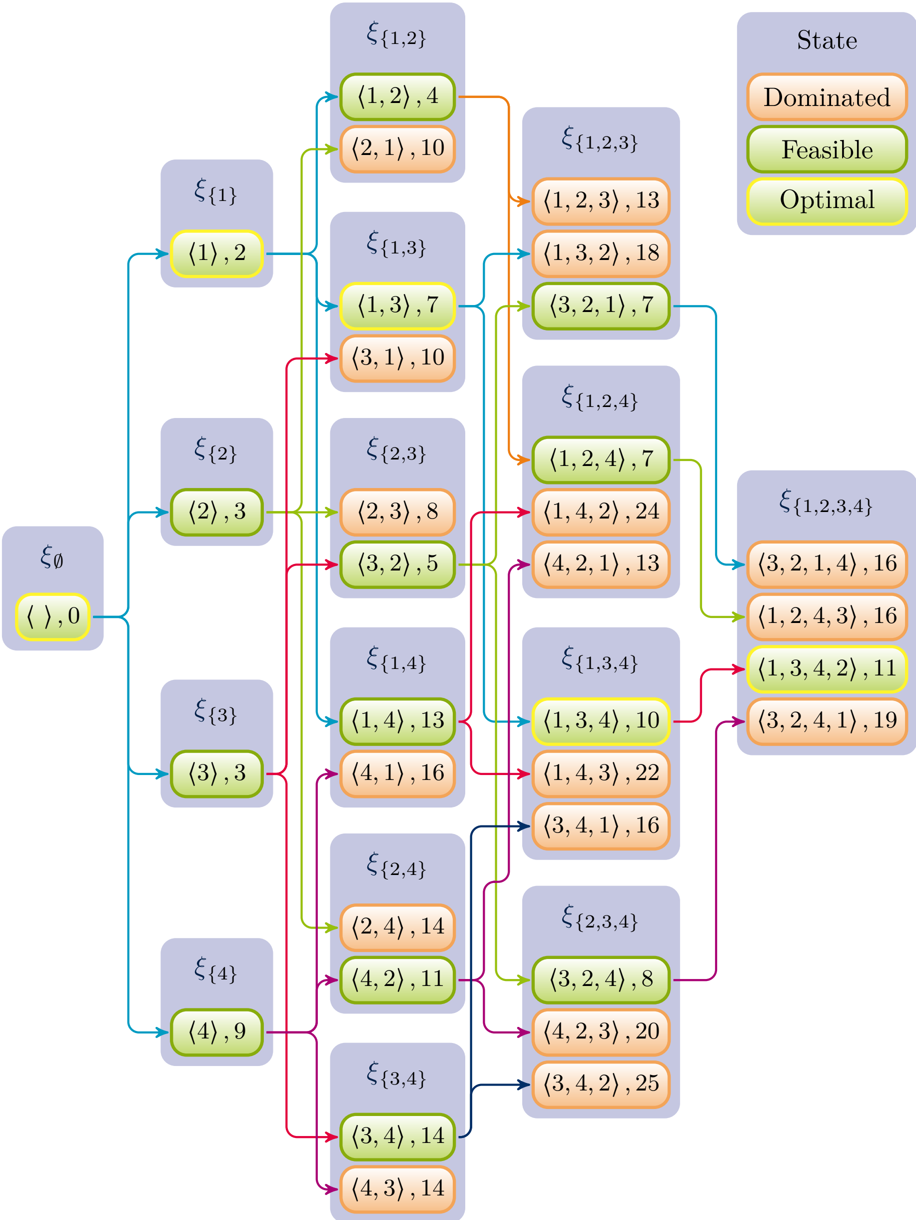


Figure 3.5: *State space of DP for the linear assignment problem in table 3.3*

3

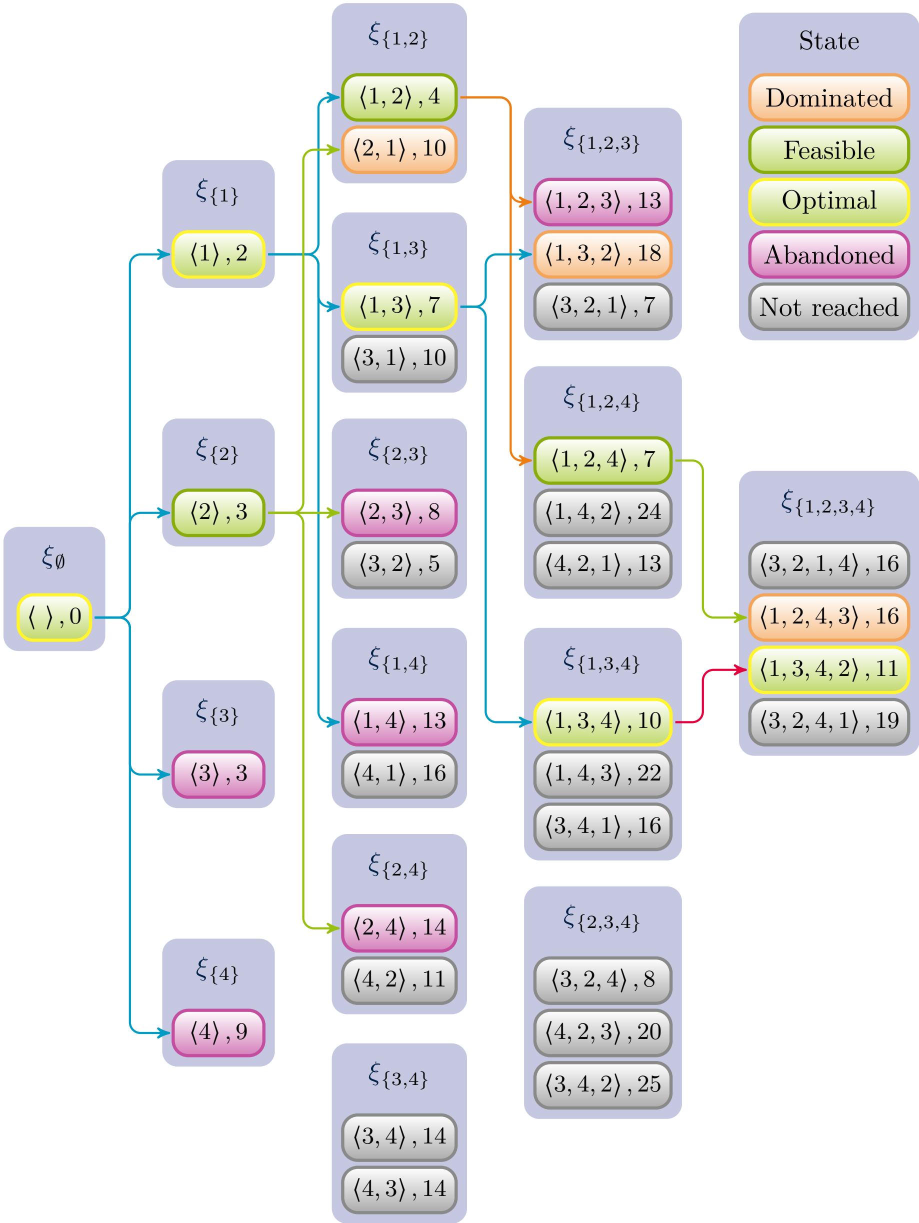


Figure 3.6: *State space of DP for the linear assignment problem in table 3.3, with $H = 2$*

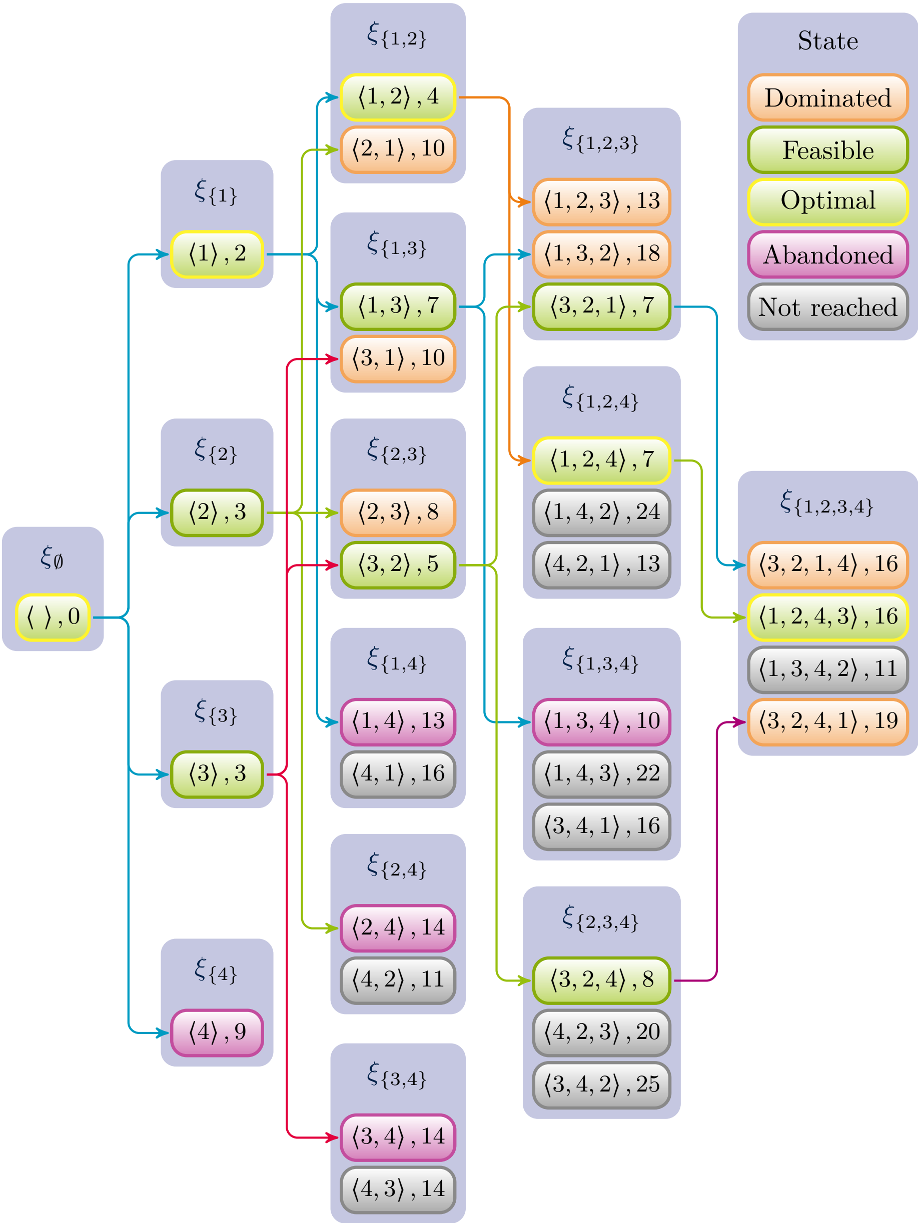
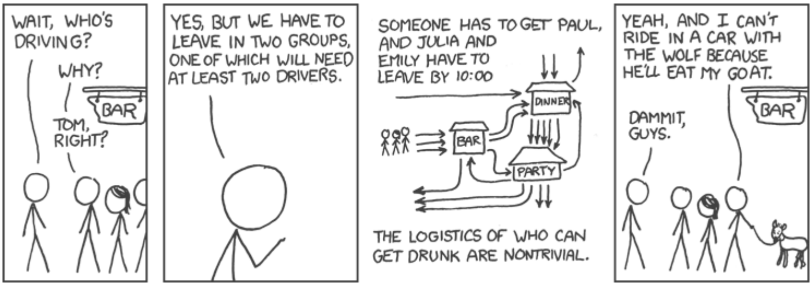


Figure 3.7: *State space of DP for the linear assignment problem in table 3.3, with $H = 3$*

to bound solutions, possibly good solutions can be considered bounded when this value turns out to be invalid as lower bound. If this estimate is sufficiently accurate it can be used to bound bad solutions, possibly using a slightly higher upper bound to prevent bounding good solutions with an incorrect estimated completion. Also, when bounding the number of solutions to be expanded by width H, this estimation of their best completion can be used to select the best H partial solutions to be expanded in each stage.



FOUR

The Vehicle Routing Problem

In this chapter we take a closer look at the Vehicle Routing Problem. First, we focus on the effects of bounding on the DP algorithm. To show these effects we use computational results on well-known CVRP instances. Second, we look at general effects of some basic properties of different VRP variants. Third, we show how to incorporate a large variety of different extensions of the VRP optimally into the VRP algorithm. Finally, we show how the DP algorithm can be used as a pricing instrument for solving rich VRPs by column generation.

4.1 Dynamic bounding for the VRP

To demonstrate the effects of bounding we tested DP on 109 instances the Capacitated Vehicle Routing Problem, see section 4.3.2. For each partial solution we calculated a lower bound on the cost of any possible completion. We ran DP with very limited state spaces to show what such state space with bounding can achieve.

We used sets A, B and P from Augerat [9], sets E, F, M from Christofides and Eilon [31], Fisher [45] and Christofides, Mingozzi, and Toth [32]. Furthermore, one large instance *G-n262-k25* from Gillett and Johnson [52] and converted TSPLIB [103] instances. All instances can be downloaded at [101].

To create an estimate on the cost of the best completion of each partial solution we used the Linear Programming (LP) relaxation, with added flow inequalities, of the Two-Commodity Network Flow Formulation of Baldacci, Hadjiconstantinou, and Mingozzi [12], where we fixed all edges already decided in the partial solution. Using the result of the LP we obtain a lower bound on the cost of any completion for each partial solution. Using this estimate on the completion we can eliminate any partial solution with an estimated completion above a given upper bound for the cost of the optimal solution. While this may eliminate some partial solutions, the greatest effect of the estimate on the completion cost is that this estimate can be used to determine the partial solutions that are to be expanded to the following stage, when the number of expanded solutions is limited by stage width H. The partial solutions chosen to be expanded we determined by selecting the H solutions with the lowest estimate on the total cost of any completion. Also, the number of feasible expansions from each node we bounded by E. All expansions to unvisited nodes are created in order of increasing order of the distances from the last node in the partial solution until E feasible expansions are found.

We ran DP for 109 instances 4 times with increasing values for the state space limitations using the cost of the previously found solution minus 1 as an upper bound. Also, the following runs are skipped when a solution with a known optimal value is obtained, since optimality cannot be proven with a pruned state space. Note that, repeating DP with an improved upper bound has limited effects on the resulting solution. As the quality of the estimated completion cost is not affected by the given upper bound, the only effect of a better upper bound on the outcome of the DP algorithm is the pruning of partial solutions for which the estimated completion cost becomes higher than the given upper bound. Typically, this estimate exceeds the given bound for small (i.e., including a small number of visited nodes) partial solutions only on very bad partial solutions, which typically do not belong to the partial solutions with the H best estimated completion costs. For larger partial solutions this can happen more frequently, thereby increasing a possibility of finding a better solution. However, increasing the size of the state space is much more effective.

We ran the following combinations for H and E: $\{H = 10, E = 10\}$, $\{H = 25, E = 25\}$, $\{H = 50, E = 25\}$, $\{H = 75, E = 25\}$. As we want to show the effect of bounding on the DP state space we report only the running time without the running cost used for bounding. The time reported is between $2 - 15\%$ of the total running time. In fact, for 80% of the runs the calculation of the LP solutions take $90 - 95\%$ of the run time, in total it takes 94.4% of the total running time of all runs. The times reported contain also the time taken for keeping the basis and fixed variables of the LP solutions to be able to hot start the calculation of each LP for consecutive partial solutions, that is a sequence of partial solutions each being an expansion of the previous partial solution. Since the calculation of each LP takes a considerate amount of running time this is not the best way to achieve a lower bound but we use it to show the effects of bounding. Part of this remaining running time is used to keep track of all LP related variables for each partial solution.

If we look at figure 4.1 which depicts the gap of the found solutions with the best known solutions we see that for more than 20%, 23, of the 109 instances an optimal solution is found. We also see that for 35% of the instances a solution is found within 1% of the best known solution, and 60% within 2%. For more than 90% of the instance we found solutions within 5% of the best known solution and for all instances we found a solution within 10% of the best known solution.

A more detailed overview of these runs can be found in table A.1 on pages 126–129 in appendix A. Also, the specifics of the machine and LP solver used to obtain these results can be found in appendix A.

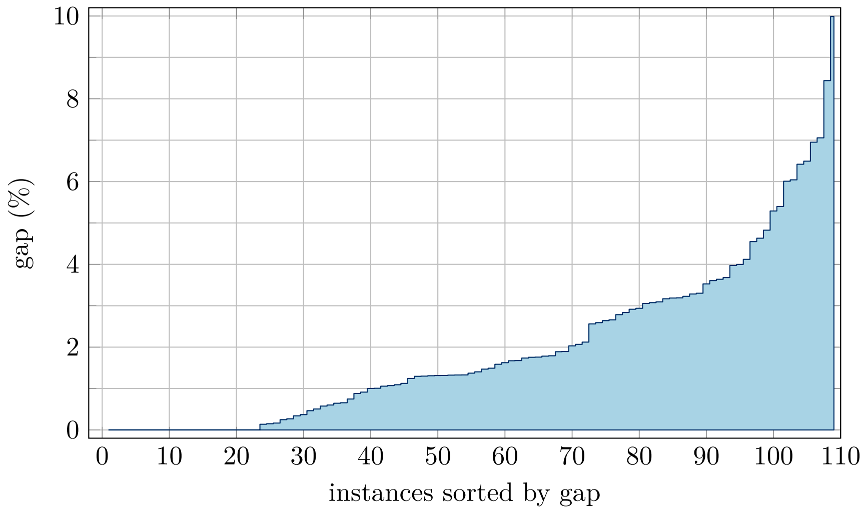


Figure 4.1: *The gap with the best known solution for the 109 instances*

4.2 Properties of the Vehicle Routing Problem

In this section we describe the effects of general properties of the VRP and TSP that are common on some variants. The precedence relations in section 4.2.1 can be the result of special constraints or just due to the possibilities of symmetry reduction given by a homogenous fleet as described in section 4.2.3. The symmetry described in section 4.2.2 applies mainly to the pure TSP and is of a more theoretical value for the VRP.

4.2.1 Precedence relations

For some problems there exist precedence relations between nodes i, j in the set V such that node i should be before node j in any feasible solution ς defined by a sequence of nodes. Such relations can have a significant impact on the running time of the DP algorithm over the set V. For any precedence relation the feasibility of an expansion can easily be checked, since all predecessors of the expanding node should be in the set S of nodes already in the solution. In this section we find the reduction of complexity of a DP algorithm over a set V when there exist different sets of precedence relations.

For the basic VRP, with homogeneous vehicles and no relations between the different customers, the order of the different vehicles in the GTR is irrelevant. The order of the vehicles can thus be chosen in advance. This can be modeled by adding precedence relations between the nodes, representing the start and the end of each vehicle, as briefly mentioned in section 2.2. Note that, this does not completely eliminate symmetry, since even with this precedence relations two equal solutions to the VRP can still have a different representation in the

GTR, as the routes of two identical vehicles can be swapped.

Let us start with the simplest precedence relation, that of just two nodes. As an example we can think of a GTR consisting of a single vehicle, where we have the precedence relation between the start at the depot and the finish at the depot of that vehicle. Let nodes $a, b \in V$ and let $a \prec b$, that is node a has to precede node b is any sequence that represents a feasible solution. For the DP state space this means that for the total set of nodes V with $n = |V|$ we have that for each subset $S \subset V$ with $b \in S$ and $a \notin S$ there exist no feasible solution. As there are $\sum_{k=1}^{n-1} \binom{n-2}{k-1} = 2^{n-2}$ of such subsets, $\frac{1}{4}$ of all possible subsets does not have any feasible solution. This reduces the time complexity of the DP algorithm by a factor of $\frac{4}{3}$.

Let us now consider a sequential precedence relation $a_1 \prec a_2 \prec \ldots \prec a_m$, which is typical for the precedence relation between the start/end nodes of the vehicles in the GTR. To count the subsets $S \subset V$ which can hold feasible partial solutions, we first look at the number of such subsets of size k, i.e., $|S| = k$. Such subset is only feasible when it contains only the first l nodes of the precedence relation, for a $l \geq 0$. The number of such subsets for a given k and l, $\{ a_1, a_2, \ldots a_l \} \subset S$ and $\{ a_{l+1}, a_{l+2}, \ldots a_m \} \cap S = \emptyset$, is equal to $\binom{n-m}{k-l}$. When we sum this over the complete state space we get $\sum_{l=0}^{m} \sum_{k=l}^{n-m+l} \binom{n-m}{k-l} = (m+1)2^{n-m}$ states that can possibly hold a feasible partial solution. If we divide this by the total number of subsets 2^n we get that a fraction of $\frac{(m+1)2^{n-m}}{2^n} = \frac{m+1}{2^m}$ of all subsets can possibly hold feasible partial solutions. So the complexity is reduced by a factor $\frac{2^m}{m+1}$ for each sequential precedence relation of length m. This result also can be achieved quickly by the following observation. Let $P = \cup_{i=1}^{m} \{a_i\}$ and let $R = V \setminus P$. Then the precedence relation has no effect on nodes in R so all possible $2^{|R|} = 2^{n-m}$ subsets are feasible. For P we only have $m + 1$ feasible subsets that are of the form $\cup_{i=1}^{k} \{a_i\}$ for $k = 0, 1, \ldots, m$. Since each feasible subset S is a union of a feasible subset of P and any subset of R, the number of feasible subsets $(m + 1)2^{n-m}$, and the fraction $\frac{m+1}{2^m}$, follows directly. In general the reduction by a related set of predecessors P is $\frac{f(P)}{2^{|P|}}$ where $f(P)$ are the number of feasible subsets of P.

With this observation we have to find $f(P)$ for each set of related predecessors to find the reduction of that set of predecessors. First we take a look at a few straightforward sets of predecessors before constructing a general expression for $f(P)$. Let the set of predecessors be $P = \{ a, b, c \}$, and let $a \prec b$ and $a \prec c$. For $a \notin S$ we have only the empty set as a feasible subset of P and for $a \in S$ we can have all 2^2 possibilities for b and c. Thus, the DP state space is reduced by a factor $\frac{8}{5}$. Similar arguments for $a \prec c$ and $b \prec c$ leads to the same factor.

For $P = \{ a, b, c, d \}$ with $a \prec b \prec d$ and $a \prec c \prec d$ we get that for $d \in S$ $P \subseteq S$, for b and c we have again all 4 possibilities given $a \in S$, when $a \notin S$ $P \cap S = \emptyset$. This leads to a reduction by a factor $\frac{2^4}{6}$.

In general we can group all nodes with exactly the same predecessors and successors, in the previous example nodes b and c could be grouped. With a sequence of precedences indirect predecessors may be removed from the set of predecessors, so if $a \prec b \prec c$ node a would not be considered as a predecessor of node c. For such a group of size g all $\sum_{k=0}^{g} \binom{g}{k} = 2^g$ possibilities are valid as soon as all predecessors are already in set S. When one of the predecessors is not in set S only a single possibility is feasible, i.e., no element of the group is in S.

Let a set of connected predecessors P be divided in q distinct groups of predecessors $P_1, P_2, \ldots, P_q$, $P = \bigcup_{i=1}^{q} P_i$ and $P_i \cap P_j = \emptyset$ if $i \neq j$, such that each element in a group P_i has equal predecessors and successors. By the definition of the groups all precedence relations in P can be expressed by precedence relations between the groups, denoted by $P_i \prec P_j$. Let the size of a group be defined by $g_i = |P_i|$ and let $h_i \subset \{1, 2, \ldots, q\}$ be the set of indices j such that $P_j \prec P_i$. Now the total number of feasible subsets of P can be calculated by a series of sums. For each group P_i we have

$$\sum_{k_i=0}^{g_i \prod_{j \in h_i} \delta_{g_j k_j}} \binom{g_i}{k_i}$$

where k_i defines the index of summation for the summation belonging to group P_i and $\delta_{g_j k_j}$ is the Kronecker delta which is defined to be 1 when $g_j = k_j$ and 0 otherwise. The product $\prod_{j \in h_i} \delta_{g_j k_j}$ evaluates to 1 when all predecessors are selected and it evaluates to 0 otherwise. The summations need of course be ordered so that all summations belonging to a predecessor occur earlier in the series, otherwise k_j would not be defined.

Let us look at a small example. Let the set of 15 connected predecessors P be divided into 5 groups, $P = P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5$ with sizes $1, 2, 3, 4, 5$, respectively. Let the precedences between these groups be $P_5 \prec P_2 \prec P_4$, $P_5 \prec P_3 \prec P_4$ and $P_3 \prec P_1$, see figure 4.2.

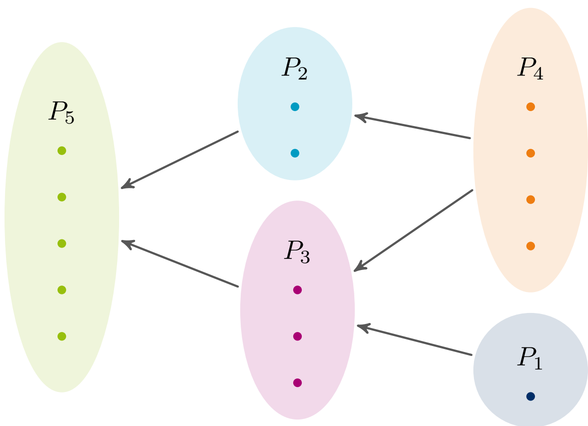


Figure 4.2: *The precedence relations in P*

Then the total number of feasible subsets of P becomes:

$$\sum_{k_5=0}^{5} \sum_{k_2=0}^{2\delta_{5k_5}} \sum_{k_3=0}^{3\delta_{5k_5}} \sum_{k_4=0}^{4\delta_{2k_2}\delta_{3k_3}} \sum_{k_1=0}^{\delta_{3k_3}} \binom{5}{k_5}\binom{2}{k_2}\binom{3}{k_3}\binom{4}{k_4}\binom{1}{k_1} = 97.$$

So, given the size $|P| = 2^{15}$ of P, the precedence relations defined in P give a reduction by a factor $\frac{2^{15}}{97}$.

4.2.2 Symmetric distance matrix

For the pure TSP with a symmetric distance matrix the computation time of the DP algorithm can be halved by a simple observation [see 40, chap. 5.3.]. In stage $\frac{n}{2}$ two states with partial solutions $\xi_{S,i \wr c}$ and $\xi_{S',i \wr c'}$ both of half a tour can be combined to a full tour with cost $c + c'$ when they are disjunct except for i, $S \cap S' = i$, and the union is equal to V except the start node s, $S \cup S' = V \setminus s$. Both tours form a route from the start node s to node i, together they cover all nodes without visiting any node twice. As the distance matrix is symmetric, each edge can be reversed with the same cost. Now one of the routes can be traversed from i to s in the opposite direction, with the same cost, forming a feasible solution to the TSP. Since both sub-routes are optimal, the resulting tour is the optimal tour which starts with S ending in i before visiting the rest of the nodes S'. All these combinations of partial solutions of stages $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ form all possible combinations of S,S' and i with $|S| = \lfloor \frac{n}{2} \rfloor$ and $|S'| = \lceil \frac{n}{2} \rceil$.

For the symmetric VRP with no extra constraints the same principle holds. However, as typically the nodes belonging to the start and end of each vehicle are in a fixed order in the GTR the comparison of S and S' should be done only in the non-depot nodes and when i is a depot node the connection can be made with any similar depot node. This reduction for the VRP is mostly of theoretical value as the solution of a pure VRP reduces to a TSP solution unless the distance matrix is non-euclidian and it is beneficial to visit the depot multiple times. Since most constraints break the symmetry, the principle fails to hold when such a constraint is added.

4.2.3 Symmetry in the GTR

For a homogeneous vehicle fleet different GTR representations can correspond to identical VRP solutions. For example, the GTR where vehicle v_1 visits only request r and the route of vehicle v_2 remains empty is identical to the GTR where vehicle v_2 visits request r and vehicle v_1 remains empty. To reduce this symmetry we can add an extra constraint on the partial solution. Before expanding to the i-th destination node, at least $\left\lceil \frac{i|R|}{m} \right\rceil$ request nodes should be in the partial solution, and for that matter in S. This enforces a non-increasing number of visited request nodes in the consecutive vehicles. For a heterogeneous vehicle fleet the same principle can be applied for each range of identical vehicles. To be able to test the feasibility for these solutions extra bookkeeping variables may be needed in the state definition, e.g., the number of customers in the last identical vehicle. We can impose these constraints as this symmetric solution should also be present in the DP state space as long as it is non-dominated. When other constraints are added other more practical tie-breaking constraints to reduce the symmetry can be added, such as a similar fraction of the total demand.

The running time implications of these constraints are hard to quantify as they not necessarily eliminate states, only with certainty eliminate expansions. Furthermore, for the VRP other beneficial effects can occur in the state space when adding constraints. For example, when the number of request nodes per vehicle are limited by a constraint, such as a capacity constraint, certain states will not contain any feasible solution. For instance, a state where S contains almost all customer nodes and has still the first vehicle as current vehicle will seldom contain a feasible solution. This effect depends heavily on the instance that is to be solved, however, in most instances of the VRP it will have some effect.

4.3 Variants of the Vehicle Routing Problem

In this section we describe a range of variants of the VRP. For various constraints of the VRP we describe how the DP algorithm can be changed to solve the VRP variant with this constraint optimally and what the complexity implications can be for the DP algorithm. These constraints can apply in similar way to the TSP, when such constraint can be applied to a single vehicle.

In general, extra constraints can be incorporated by adding extra state variables which, in general, have negative impact on the performance. In the worst case this even can lead to a brute force algorithm, when every solution is the sole solution of its state. We will see that for some constraints no state variables need to be added and even that constraints can have a positive impact on the performance when the reduction in feasible solutions can be incorporated in the existing DP algorithm in a natural way.

When the DP algorithm is converted into a heuristic, the state variables added in this section are typically the variables that are removed as described in section 3.3.1.

4.3.1 Heterogeneous vehicles

For the VRP it is possible that certain vehicles have different characteristics. The most common reasons for heterogeneity are capacity and travel speed as we will see in sections 4.3.2 and 4.3.6. However, the simplest form of a heterogeneous vehicle fleet is a constraint which forbids for a customer to be visited by certain vehicles. A few practical examples of this constraint are frozen goods that may only be transported by vehicles equipped with a refrigerator, a certificate needed by the driver of the vehicle, for example to transport hazardous goods, or the availability of a crane on the vehicle. Another simple example of a heterogenous vehicle fleet are different origin and destination locations for each vehicle. Also, the Open-VRP, vehicles do not have to return to the depot, can be modeled by setting the destination location at distance 0 from each origin and request node. The same principle holds for the Closed-Open-VRP where only the distance to the destination nodes from the "Open" vehicles is set to 0. To be able to handle such constraints we can use the same state space definition.

From the set S we can deduce the current vehicle in the GTR, as this is the only vehicle for which the origin node is in set S while the destination node is not in set S. Even when the order of the vehicles is fixed, and pairs of the destination node and the origin node of two consecutive vehicles are merged into a single node, the current vehicle can be found, this can be done by finding what is the latest of such nodes in S. This can easily be found as they should be added to S according to their fixed precedence relation.

This constraint has a favorable effect on the running time of the DP algorithm, as certain expansions are forbidden and states $\xi_{S,i}$ where i may not be loaded into the current vehicle defined by S cannot hold feasible solutions. However, the number of states that are actually removed by this constraint depends on the instance. Even if we take the effort in calculating the number of sets S for which state $\xi_{S,i}$ has no feasible solutions, it is very likely that the effect described in the second paragraph of section 4.2.3 will have a higher impact on the running time of the algorithm.

4.3.2 Capacitated VRP

To solve the Capacitated Vehicle Routing Problem (CVRP) by a DP algorithm we have to add an extra variable to the state definition. We need this variable to check if the maximal capacity of a vehicle is not exceeded as well as to be able to check the domination between two solutions in the same state. A solution $\varsigma_{S,i\wr c}$ may have a higher cost compared to an other solution $\varsigma'_{S,i\wr c'}$ in the same state ($c > c'$), however, solution $\varsigma_{S,i\wr c}$ can have more slack for the demand, that is, have less capacity used, then solution $\varsigma'_{S,i\wr c'}$. This breaks the optimality principle, since not all completions of $\varsigma_{S,i\wr c}$ have to be feasible completions of $\varsigma'_{S,i\wr c'}$.

To restore the optimality principle we add an extra state variable q, depicting the remaining capacity, to the array γ of a state variable we compare within a state. We are able to add q to γ instead of ϕ, since a solution with a lower cost and a higher slack in demand dominates another solution for the same set S and terminal node i. The new state definition becomes $\xi_{S,i\wr c,q}$ where $\gamma \geqq \gamma'$ is defined as $\geqq = \{\leq, \geq\}$. We are able to update q correctly as the current vehicle, as well as its start at the depot, can be determined by the set S. At such a node we can set q to Q_j, where Q_j is defined as the capacity of vehicle j. At each request node the demand at that node can be subtracted from the state variable q leaving the correct remaining capacity. A partial solution is infeasible due to the capacity if and only if $q < 0$, surpassing the remaining capacity. As we can see, instances with a vehicle fleet that is heterogeneous by capacity can perfectly be solved by this extended DP algorithm.

The addition of the capacity constraints multiplies the theoretical running time for the DP algorithm for the VRP by a factor $Q + 1$. Here, $Q = \max_{j\in V} Q_j$ is the maximal capacity of any vehicle. For each value of $0 \leq q \leq Q$ we can have a non-dominated solution $\varsigma_{S,i\wr c,q}$ in state $\xi_{S,i}$. For example by having increasing values for the costs, $c = M + q$ with some constant M. This leads to a time complexity for the DP algorithm for the CVRP of $\mathcal{O}\big(Q\,(n+m)^2\,m2^n\big)$. However, in practice not all values will occur and solutions with different remaining capacity will dominate each other. This is the same principle as we have seen with the estimate by anti-chains as described in section 2.3.

Note: for a homogeneous vehicle fleet it may be profitable to break the symmetry of the GTR in a bit different way than described in section 4.2.3 by using a fraction of the demand instead of the number of request nodes as extra constraint before allowing a vehicle to return to the depot. Especially when it is to be expected that there is more spread in total demand delivered by each vehicle than the number of request nodes visited by each vehicle.

4.3.3 Multiple compartment VRP

The CVRP can be extended to the Multiple Compartment Vehicle Routing Problem (MC-VRP) [28,42,105] by adding compartments, each with their own capacity, to each vehicle. A simple example of a vehicle with multiple compartments can be a truck with two or more trailers, or a truck with a cooling section. For each vehicle v_j we have a number of compartments p_j and for each compartment p_j^k a maximum capacity Q_j^k ($j \in \{1,2,\ldots,m\}$, $k \in \{1,2,\ldots,p_j\}$). Furthermore, for each customer-vehicle combination we have a set of allowed compartments P_{ij}, where the complete demand has to be loaded into a single compartment. Customers where the demand has to be delivered from multiple compartments can be modeled by multiple customers at the same location. When it is required that such demands are delivered consecutively it is easy to add a fixed precedence relation between these virtual customers and to add a constraint that they will need to be visited consecutively. It is possible to combine them into a single customer request, however, this will explode the number of expansions, and is from now on disregarded for the sake of the simplicity of this section.

To solve the MC-VRP by a DP algorithm we have to be able to check on the remaining capacity of each compartment. To be able to perform this check we add a state variable $\vec{q}$ to γ, similarly to the state definition of the CVRP, representing the remaining capacity of each of the compartments. Since the current vehicle is uniquely defined by S, the length of $\vec{q}$ is equal for all solutions within the same state, and the values of $\vec{q}$ correspond to the same compartments. For each expansion to a new node we have to make a choice in which compartment, of the current vehicle v_j, to load the demand. To represent this choice, we make not a single expansion for each new node i, but $|P_{ij}|$ expansions for the same node i, one for each compartment where it is allowed to load the demand q_i.

Since the new state variable $\vec{q}$ is a vector, this can lead to a lot of non-domination partial solutions within the same state, as we have seen with the JSSP in section 2.3. Let p be the maximum number of compartments in any vehicle defined by $\max_{v_j \in V} p_j$ and let Q the maximum compartment size in any vehicle defined by $Q = \max_{v_j \in V} \max_{k \in \{1,2,\ldots,p_j\}} Q_j^k$, we have at most $(Q+1)^p$ possible value combinations in the vector $\vec{q}$. Combining this with the maximal number of expansion P for a single partial solution to a single node, defined by $P = \max_{v_j \in V, i \in R} |P_{ij}|$ over all vehicle customer combinations. The time complexity for the DP algorithm for the MC-VRP becomes $\mathcal{O}\big(PQ^p\,(n+m)^2\,m2^n\big)$.

When there is no constraint on the compartments, that is for each customer-vehicle combination all or none compartments are feasible ($|P_{ij}| \in \{0, p_j\}$). Then just the remainders of all compartments have to be known to check the feasibility, not which compartment has which remaining capacity. This fact can be used to reduce the number of possible non-dominated solutions per state by sorting $\vec{q}$ on the remaining capacity. For example, when $\vec{q}$ is sorted, it can be that two expansions, from the same solution to the same node for different compartments, will be considered as equal within their state, so only one will remain. This occurs when the demand is placed in two different compartments with originally the same remaining capacity.

4.3.4 Pickup and delivery

For the Vehicle Routing Problem with Pickup and Delivery (VRPPD) the demands are not solely pickup or solely delivery, as they are for the CVRP, so a simple capacity check is not sufficient. The VRPPD combines linehauls from the depot (deliveries) and backhauls to the depot (pickups). Typically, no goods are transported between two customers. Exchanging goods between customers and matched pickup and delivery where a specific good needs to be transported from location a to location b are discussed later in this section.

For the VRPPD it is no longer sufficient to check the capacity at each customer using only the remaining capacity. For the CVRP, the capacity is checked using this remaining capacity. In the case of backhauls, the reaming capacity represents the actual remaining capacity. In case of linehauls, the vehicle is "empty" after planning each customer, since a delivery is just made and the reaming capacity represents the capacity that still can be loaded at the depot. When we combine these two flavors we have to keep track at the capacity we are still able to load at the depot and the capacity we have to transport new load to the depot. We simply change the state definition $\xi_{S,i\wr c,q}$ of the CVRP to $\xi_{S,i\wr c,q_d,q_p}$. Here, q_d is the capacity we are still able to deliver at this point in the route of the current vehicle. That is, to be able to transport this from the depot to the current node in the route without violating the capacity of the vehicle anywhere along the route. The last variable q_p is the capacity left to transport load from the current node to the depot at the end of the route, this thereby also represents the remaining capacity in the current vehicle after node i.

Both q_d and q_p start at the vehicle capacity Q_j of vehicle j at the start of the route of vehicle j. When the current node is a delivery node i_d the demand of that node is subtracted from q_d and q_p is left the same as the same amount can be loaded after node i_d. When the current node is a pickup node i_p the demand of this node is subtracted from q_p as this load will occupy the vehicle until the depot is reached. Furthermore, we set $q_d = \min\{q_d, q_p\}$ as the amount we can transport from the depot may be limited by what we already have loaded during the route until the current customer. A partial solution is infeasible when $q_d < 0$ or $q_p < 0$, since this means that the remaining capacity is insufficient at the current node (q_p) or somewhere earlier along the route (q_d).

The extra variable in the state definition adds an extra factor Q, number of distinct values that can be obtained by q_p, in the time complexity of the DP algorithm in comparison to the CVRP, it now becomes $\mathcal{O}\big(Q^2\,(n+m)^2\,m2^n\big)$.

For a matched pickup and delivery problem where a specific good needs to be transported from location a to location b for each customer we add nodes for each a and each b. A precedence relation is to be added between each pair a, b. Also we have to make sure that when a vehicle visits a it also visits b, for this a feasibility check can be added to ensure a vehicle returns only to the depot when for each pair a and b holds that either none or both are scheduled. This can easily be checked using only the set S in the state definition.

For customers where the demand is so high that no other demand can fit together with this demand in any of the vehicles, the so-called Full Truck Loads, the pickup and delivery node in the DP algorithm can trivially be contracted into a single node that starts at the pickup location and finishes at the delivery location, reducing the number of nodes.

The effect on the time complexity described above is that n may be replaced with $2n$ depending on how the original problem is described. Furthermore, extra precedence relations are added giving reductions described in section 4.2.1. For k paired precedence relations this gives a reduction by a factor of $\left(\frac{4}{3}\right)^k$.

4.3.5 Redistribution

Sometimes it is allowed to deliver goods that are not loaded at the depot but at other customers. A good example for this variant is the redistribution of bicycles over different rental locations in a city where it is allowed to return a bike at another location than where it is rented. For this Pickup and Delivery variant the state definition does not change. Instead we change the way q_d and q_p are calculated. The process at the pickup node stays identical while at the delivery node first is checked how much load is present in the current vehicle $(Q_j - q_p)$ to fulfill the demand at the delivery node. The demand that can be fulfilled with the load in the vehicle is added to q_p as it is unloaded and only the part of the demand that needs to be fulfilled by loading at the depot is subtracted from q_d. The theoretical time complexity is the same as described above. However, in practice we can expect the actual running times of this variant to be higher than the running times of the VRPPD, since the value of q_p is not strictly decreasing. This can lead to more non-dominated partial solutions per state.

When there are more types of load transported, at each location multiple types can be delivered or picked-up, or even a combination of the two. In the previous example there may be three types of bicycles, man, woman and kids. In that case q_p can be turned into a vector $\vec{q}_p$ where it represents not the capacity that can be loaded but the actual loaded amount. So a partial solution renders infeasible when $\sum \vec{q}_p > Q$. Now the actual load in the current vehicle is available and the corresponding checks at a delivery can be made. For this extension the time complexity of the CVRP could be multiplied by $Q^{|\vec{q}_p|}$, as all entries of q_p can range from 0 to Q. However, the values of $\vec{q}_p$ are limited since $\sum \vec{q}_p > Q$.

This gives at most $\binom{Q+|\vec{q}_p|}{|\vec{q}_p|} = \mathcal{O}\left(\frac{Q^{|\vec{q}_p|}}{|\vec{q}_p|!}\right)$ possible values in $\vec{q}_p$. This gives a time complexity of $\mathcal{O}\left(\frac{Q^{1+|\vec{q}_p|}}{|\vec{q}_p|!}(n+m)^2 m2^n\right)$.

Also, a limited capacity at the depot can be handles by adding a state variable q_D which represents the stock (of bicycles) available at the depot. This adds an extra factor Q_D to the complexity, where Q_D is the initial stock at the depot. This can also be extended to multiple types by using $\vec{q}_D$ thereby adding $\prod \vec{Q}_D$ to the time complexity.

4.3.6 Time-windows

When we add time-windows to the VRP we get the VRPTW. For the VRPTW we have to add an extra state variable t_v, even when the objective is to minimize the total time, where t_v represents the time of the current vehicle. With the TSP it may be possible to use the cost as time, since there is only one vehicle. Service times at each customer can be incorporated trivially as this just changes the value of t_v after the expansion. The feasibility can easily be checked with state-variable t_v and t_v can simply be calculated for each consecutive partial solution. This adds an extra factor of $t_{\max}$ to the time complexity, where $t_{\max}$ is the maximal allowed time for any vehicle.

Since the current vehicle can be deduced from the set S, variable travel times for each vehicle can easily be incorporated without any difference in the time complexity. This can be done by a vehicle dependent factor on the travel-time/distance matrix. In fact any function depending on the current vehicle, the edge and the departure time can be used. The only condition for the function is that the arrival time should be monotonic in the departure time. This prevents that departing later could result in a earlier arrival time. However, when such a function is needed extra waiting time could be added to find the best departure time to get the earliest arrival possible according to the current earliest departure time. This allows for time dependant travel times to take (expected) congestions into account.

4.3.7 Driving and working hours regulations

Also driving and working hours regulations for routes in the VRP can be incorporated in the DP algorithm. Driving and working hours regulations are enforced in many countries to limit the driving and working hours of truck drivers to prevent accidents due to driver fatigue. These regulations are often defined in terms of total working and driving hours per day, and the total of consecutive driving hours before a short break is obligatory. These rules are often defined with complicated exceptions like the possibility to split up breaks or shorten a daily rest a few times a week. More specifics on these rules can be found in Goel [53,54] and Goel and Kok [55]. Much more on this specific topic can be found in Kok.

Goel [54] describes a labeling algorithm to find a feasible solution for a route according to the European Union driving hours regulation. The labeling algorithm uses different activities like DRIVE, WORK, BREAK and REST to be sequenced for a given sequence of customers. This labeling can be directly incorporated into the DP algorithm where all the labels that are used for domination can be added to γ. Two domination criteria of Goel [54] consist of two labels which can be named and also be added to γ. For each expansion in the DP algorithm all possible choices in the labeling algorithm between two customers need to be considered as an expansion. This is similar to the multiple expansions described in section 4.3.3.

Adding this labeling algorithm to the DP algorithm simply adds one, albeit large, constant factor to the time complexity for a given set of regulations. For a given set of regulations the number of non dominating labels for a given sequence and the number of possible options of activities between two customers is limited by a constant.

4.3.8 Inter-route time constraints

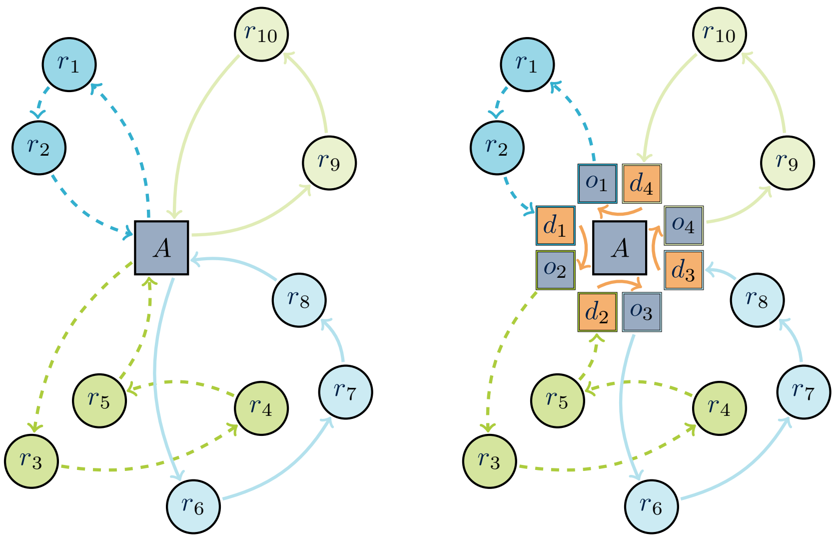
Sometimes it is allowed for a vehicle to visit the depot multiple times. To allow for this in the DP algorithm, when there are not yet separate nodes for the pickup and the delivery, extra nodes at the depot can be created. These can be seen as customers at the depot with zero demand that have to be handled by specific vehicle in a specific order. At such a customer the remaining capacity is reset to the full capacity of the vehicle.

4

It is also possible to model each route of a vehicle as a single tour in the GTR. This creates precedence relations for the start and finish nodes of these routes, it also creates inter-route dependencies for these tours for which extra state dimensions have to be added. However, it also allows us to alternate the routes of different vehicles. For example the VRP solution in figure 4.3a where we have two routes for each of the two vehicles blue and green. For each of the two vehicles the lighter route has to be performed after the darker (dashed) route. A possible GTR for this solution is given in figure 4.3b where the first route of each vehicle is performed first while the vehicles are alternated in the GTR. This principle allows us to cope with a wide variety of other inter-route constraints.

To be able to alternate between routes of different vehicles the current state of each vehicle have to be kept in the state variables, such as the current time and the current remaining capacity for each vehicle. As long as the nodes for each trip where a new sub tour can start are known, the last node for each vehicle can be derived from the set S. Naturally, adding all these state variables explodes the time complexity of the DP algorithm.

With these multiple routes extra inter-route constraints become available to incorporate into the DP algorithm. For example the possibility to let drivers to interchange trucks or trailers at some location, for example a country border, to let them return home twice as fast. This can be modeled by having two routes for each vehicle with precedence relations between the routes of the different



a: *A VRP solution*

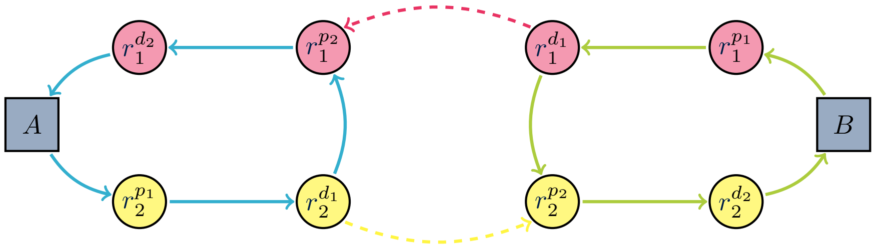
b: *A corresponding GTR*

Figure 4.3: *An example of a VRP solution with 2 vehicles with each two routes*

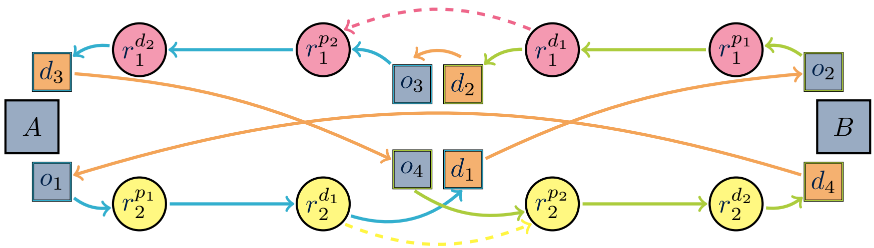
4

vehicles. For example in figure 4.4 such a solution with its corresponding GTR is given. Notice the order of the routes in the GTR, this order has to respect the order of the transports, given in red and yellow, as well as the order within each of the vehicles. This order is not fixed within the GTR over all solutions, the solutions dictate the feasible orders of the routes within the GTR. The waiting time at the border can be modeled by setting the current time of the first vehicle to arrive to the current time of the second vehicle at its arrival. Due to the precedence relations the first vehicle is prohibited to leave before this arrival. Note that the choice of the pairs of drivers interchanging their equipment is not made by this model. The locations of the origin and destination nodes between multiple routes of the same vehicle can be variable in the model by setting all distances of these nodes to zero.

Another example is the combination of long-haul routes with fine distribution in a single VRP. A good example for this kind of planning is parcel delivery. Packages are retrieved at the customers and shipped to the closest warehouse. From there truckloads are combined to be transported between warehouses. The final delivery is made again from the warehouse closest to the delivery location. For a single parcel we have three different pickup and delivery pairs that have time and thereby precedence relations between them. To be able to track the different parcels we have to add the time they are available to ship from the warehouse into the state variables. This time needs to be added for each parcel that is in the current GTR in a warehouse, that is for each parcel where at least one pickup and delivery pair is scheduled (in S), but not all of them. The resulting relations are similar to the relations created by the two transports in



a: *A VRP solution*



b: *A corresponding GTR*

Figure 4.4: *An example of a VRP solution where trailers are switched at the border*

figure 4.3 between the first route of each vehicle and the second route of the other vehicle.

4.3.9 Mixing variants

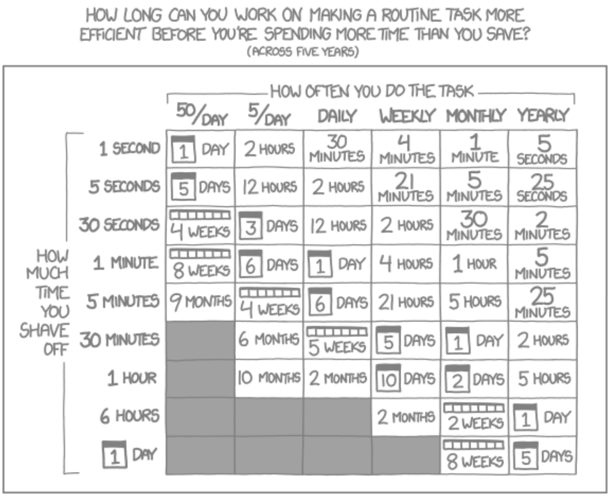
It is in general possible to mix the different variants such as explained in this section. This makes DP a general framework for both exact and heuristic VRP solutions. The importance of this fact may become more apparent in the next section.

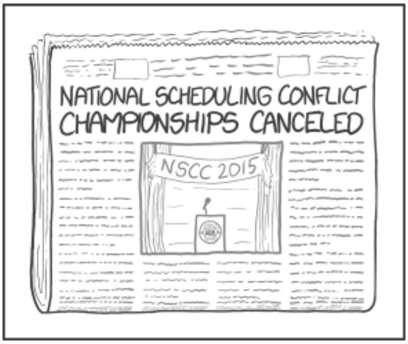
4.4 Using DP as pricing instrument

A state space can grow beyond practical usage when trying to solve a VRP by DP, especially when extra state variables need to be added. Bounding and using DP as a heuristic can limit the state space but may have adverse effects on the running time performance and solution quality.

To limit the size of the DP algorithm that has to be calculated a slightly altered version of our DP algorithm can be incorporated into a column generation technique as a pricing instrument. We did not test such a procedure, but we want to mention it here as it may be promising to use our general framework for solving rich VRPs as pricing instrument within a column generation framework. For a general introduction to column generation for the VRP we refer to Feillet [43]. In Dell'Amico, Righini, and Salani [35] a DP algorithm is used as a pricing instrument. A large variety of pricing routines can easily be achieved from our DP framework. Since the DP algorithm should only create a route for a single vehicle the DP state space becomes more limited. Only nodes for a single vehicle are needed and the number of request nodes that can be planned efficiently into a single vehicle is almost always very limited, especially for rich VRPs.

However, the algorithm should slightly be altered. The DP algorithm described for the TSP and VRP in this dissertation gives only solutions where all requests are planned. Implicitly, the last node of the last vehicle could only be planned when all requests are planned otherwise every expansion on such solution would be infeasible and the last stage would not be reached. When DP is used as pricing instrument the node that represents the end of the route, typically the return to the depot, may be scheduled as expansion of any partial solution, provided that this expansion is feasible. The main difference with the previous described algorithm is that such an expansion was a-priori infeasible, while as pricing instrument such infeasibility can only occur as effect of the specific type of VRP. For example, with a matched pickup delivery problem, see section 4.3.4, the return node of a vehicle can only be scheduled when all deliveries matching the pickups are scheduled. Finally, every partial solution where the return node of the vehicle is scheduled should be considered as final result. This is similar to the principle for the Steiner tree problem described in section 1.2.2.





FIVE

The Job Shop Scheduling Problem

In this chapter we look in more detail at the Job Shop Scheduling Problem. We show a way of adding bounding to the DP algorithm for the JSSP. As the algorithm for the bounding described in section 5.1 is an implementation of existing work, the description given is brief. The description of the bounding algorithm with maintenances in section 6.4 is more elaborate. This description can also be used to get a better understanding of the specifics of the bounding algorithm for the general JSSP, instead of the references given in section 5.1. With computational results we show what the effects of these bounds can be on the performance of the DP algorithm. Using dynamic bounding also lower bounds for an instance can be found, we use this to improve the lower bound of 16 instances. We show in detail how to change the procedure of section 3.2 in order to find all optimal solutions to a JSSP.

5.1 Dynamic bounding for the JSSP

To limit the size of the state space of the DP algorithm bounding can be used to remove partial solutions as described in section 3.1. Many different approaches are possible when applying bounding to the DP state space. In fact, any procedure that provides a lower bound on the completion of a partial solution can be used to prune the state space.

We implemented the parallel head–tail adjustments of Brinkkötter and Brucker [21] to be used as bounding for each partial solution of the DP state space. The advantage of this algorithm for us is that the heads and tails can be saved with each partial solution to be reused when an expansion is made. Also the scheduled operations can easily be fixed by setting the head and tail to the start time and the upper bound minus the finish time of the operation, respectively. Another advantage is that this algorithm also provides precedence relations between operations which can be used by the DP algorithm. These precedences are specific for a partial solution and all extensions derived from it. This can easily be incorporated into the DP algorithm by allowing only expansions for which all predecessors are already in the partial solution.

For each expansion, we use this algorithm to recalculate the heads and tails, any new precedences are also saved. It is possible that the given upper bound is invalid for any completion of a partial solution, in which case the partial solution is regarded as infeasible. All new variables, the heads and tails as well as the precedences could be stored in the bookkeeping β variables of the state definition. However, the (only interesting) values for not yet scheduled nodes could conceptually be derived from the current aptitude value for each job and the global upper bound. As head of the first unscheduled operation for each job the aptitude value minus the operation time would be used. For performance reasons the heads, tails and precedences can be cached, but the state definition does not have to be changed.

With this bounding the running time of the DP algorithm can be drastically decreased, as can be seen in section 5.2. A valid upper bound can be obtained by any heuristic, for example a heuristic version of the DP algorithm itself can be used. To improve this bound the DP algorithm including the bounding can be run iteratively while limiting the number of partial solutions to be expanded in each stage, as long as the upper bound improves.

In the next section the parallel head–tail adjustments of Brinkkötter and Brucker [21] are described.

5

5.1.1 Parallel head–tail adjustments

The parallel head–tail adjustments of Brinkkötter and Brucker [21] are described more elaborately in Brinkkötter and Brucker [20]. This section describes briefly this algorithm and how it can be incorporated into the DP algorithm.

First we start with a single machine problem obtained by taking all operations I of one machine of the original JSSP. Now we define for each operation $o \in I$ a head r_o and tail q_o. The corresponding head–tail problem is that of finding a schedule for which each operation does not start before its head and for which maximum of the finish time plus the tail ($\max_{o \in I} \{C_o + q_o\}$) is minimal.

The *Jackson's preemptive schedule* (JPS) is an optimal solution to the preemptive version of this problem and can be obtained by applying the following procedure:

> At time t schedule an available operation with the largest tail, until either the completion time of this operation or the time defined by the smallest head with $r_o > t$.

Consider a partial schedule constructed by this procedure until time $t = r_w$ defined by the head of operation w and let p_o^+ be the remaining processing time for operation $o \in I$ at time t.

The head–tail adjustments are based on two results from Carlier and Pinson [24]. Let $\mathcal{UB}$ be an upper bound for the single machine problem, when for some operation $o \neq w$ we have that

$$r_w + p_w + p_o + q_o > \mathcal{UB}.$$

Operation w cannot start before operation o is finished and $r_w \geq r_o + p_o$. Furthermore when we have for some subset $Y \subset I$ with $Y \subseteq \{ o \in I \mid p_o^+ > 0 \} \setminus \{w\}$ that

$$r_w + p_w + \sum_{o \in Y} p_o^+ + \min_{o \in Y} q_o > \mathcal{UB}$$

holds, then w cannot start before any operation in Y. Therefore, r_w can be set to $r_w + \sum_{o \in Y} p_o^+$. Note that the first inequality is covered by the second inequality when $p_o^+ = p_o$, so it needs only to be checked if $p_o^+ \neq p_o$.

The head–tail adjustments are done for a single machine by using these two inequalities, by finding the maximal set Y efficiently, and update all the heads in a single sweep of finding the JPS. Thereafter the roles of heads and tails are reversed. In the next sweep the heads, now actually the tails, are updated again. This procedure is repeated until no further updates are possible.

For the JSSP the heads for all operations on all machines can be updated simultaneously by performing the head–tail adjustments in parallel for all machines updating the heads of all operations of the JSSP in one sweep, using the upper bound $\mathcal{UB}$ of the JSSP for all machines.

To incorporate this into the DP algorithm for any solution ς_S created by an expansion the heads of the next operations for each job ($o \in \varepsilon(S)$) can be set to $\alpha(\varsigma_S, o) - p(o)$. The precedence relations found during the parallel head–tail adjustments can be used to limit the possible expansions later in the DP state space for all descendants of solution ς_S. Solution ς_S can be discarded when no Jackson's preemptive schedule can be found within the $\mathcal{UB}$.

5.2 Finding JSSP solutions

To demonstrate the capabilities of DP on the JSSP we used several sets of benchmark instances. First we used no bounding and without limiting the state space only very small instances could be solved. With bounding added to DP many more instances could be solved, even with a very limitative state space width, and for a lot of instances optimality can be proven. We also used DP to obtain new lower bounds for 16 instances.

The 242 instances we used are from eight sets: Fisher and Thompson [44], Lawrence [79], Adams, Balas, and Zawack [2], Applegate and Cook [7], Storer, Wu, and Vaccari [110], Yamada and Nakano [123], Taillard [111] and Demirkol, Mehta, and Uzsoy [38]. A detailed overview of these instances including the best known values for their bounds with references to their origin can be found in appendix B. Specifics on the computer used to obtain these results can be found in appendix A.

To achieve a DP-based heuristic for the JSSP by limiting the state space we only limited the number of partial solutions to be expanded by setting H

(section 3.3.3) and not by limiting the number of expansions for such a partial solution. So all feasible expansions are created and the bound E (section 3.3.2) is not used as the number of feasible expansions is already limited to N, since at most a single operation per job is feasible to be scheduled next. To select the H solutions to expand for each stage, the lower bound on the completion, as described in section 5.1, is used.

5.2.1 No bound

When we set no bound the state space from the DP algorithm gets huge. Without any bounding we were able to solve and prove optimality only for instances up to just 50 operations. This takes quite some time and memory, as can be seen in table 5.1.

Instance	# Jobs	# Machines	Optimum	CPU (s)	Memory (MB)
ft06	6	6	55	0	3
la01	10	5	666	1163	4937
la02	10	5	655	1502	6163
la03	10	5	597	931	3674
la04	10	5	590	1230	5384
la05	10	5	593	726	3279

Table 5.1: *Runs with no bounds*

5

However, the computation effort observed is much less than the upper bound established in section 2.3. As we have seen in that section the maximum number of non-dominated solutions for any state is approximately bound by $U = \frac{{p_{\max}}^N}{\sqrt{N}}$. The maximum total number of solutions becomes then $\frac{(M+1)^N {p_{\max}}^N}{\sqrt{N}}$, since there are $(M+1)^N$ states possible with feasible solutions. When we compare these theoretical bounds to the results obtained from the runs above which did not limit the state space and did not use any bound, we see that the actual results are

Instance	# J	# M	Observed count		Theoretical $\mathcal{UB}$	
			per state	total	per state	total
ft06	6	6	13	30 410	$408 \cdot 10^3$	$48\,030 \cdot 10^6$
la01	10	5	142	63 170 930	$25\,838 \cdot 10^{15}$	$1562 \cdot 10^{24}$
la02	10	5	157	80 862 884	$28\,599 \cdot 10^{15}$	$1729 \cdot 10^{24}$
la03	10	5	191	50 910 277	$12\,314 \cdot 10^{15}$	$745 \cdot 10^{24}$
la04	10	5	114	68 208 803	$25\,838 \cdot 10^{15}$	$1562 \cdot 10^{24}$
la05	10	5	182	40 229 132	$23\,319 \cdot 10^{15}$	$1410 \cdot 10^{24}$

Table 5.2: *Number of partial solutions*

factors lower, see table 5.2. This suggests that it may be possible to improve our theoretical bound and hence establish that our algorithm has a lower complexity than the upper bound on the complexity we have proven.

5.2.2 Optimal bound

With the use of the bounding described in section 5.1 and allowing only 3 states per stage ($H = 3$) to be expanded, see section 3.3.3, we were still able to find the optimal solution for 14 instances when we used the optimal value as a bound. Note, this does not prove the optimality. These runs are naturally very quick, most within a second, see table 5.3. We find it very surprising to find so many optimal solutions by such a narrow search within the total state space.

Instance	# Jobs	# Machines	Optimum	CPU (s)	Memory (MB)
ft06	6	6	55	0	1
ft20	20	5	1165	0	1
la05	10	5	593	0	1
la06	15	5	926	0	1
la07	15	5	890	0	1
la08	15	5	863	0	1
la09	15	5	951	0	1
la10	15	5	958	0	1
la11	20	5	1222	0	1
la12	20	5	1039	1	1
la13	20	5	1150	0	1
la14	20	5	1292	0	1
swv16	50	10	2924	27	2
swv17	50	10	2794	24	2

Table 5.3: *Runs with $H = 3$ and bounded with the optimal value*

Without restricting the number of expansions and using the optimal values as bounds we could prove the optimality of all instances with a maximum of 10 jobs. Other instances would require more memory. With these instances the run times are very quick, except for two cases the optimality was proved within a minute, see table 5.4.

One of these instances is the famous instance *ft10* proposed in 1963 by Fisher and Thompson [44], which was first solved in 1986 by Carlier and Pinson [24]. We solve this instance in 60 seconds using 95470 non-dominated states in the complete state space. The majority of these states are in the first 20, of the 100, stages of the state space, in the last 80, 75, and 70 stages there exist only 7315, 776, and 256 non-dominated solutions, respectively. See figure 5.1 for the number of non-dominated solutions per stage for the run of *ft10* in table 5.4.

Instance	# Jobs	# Machines	Optimum	CPU (s)	Memory (MB)
abz5	10	10	1234	42	8
abz6	10	10	943	3	2
ft06	6	6	55	0	1
ft10	10	10	930	60	14
la01	10	5	666	25	12
la02	10	5	655	1	1
la03	10	5	597	0	1
la04	10	5	590	0	1
la05	10	5	593	466	243
la16	10	10	945	44	10
la17	10	10	784	1	1
la18	10	10	848	14	4
la19	10	10	842	9	2
la20	10	10	902	3	1
orb01	10	10	1059	36	8
orb02	10	10	888	29	6
orb03	10	10	1005	185	26
orb04	10	10	1005	20	6
orb05	10	10	887	29	5
orb06	10	10	1010	40	8
orb07	10	10	397	8	3
orb08	10	10	899	9	3
orb09	10	10	934	14	4
orb10	10	10	944	1	1

Table 5.4: *Runs bounded with the optimal value*

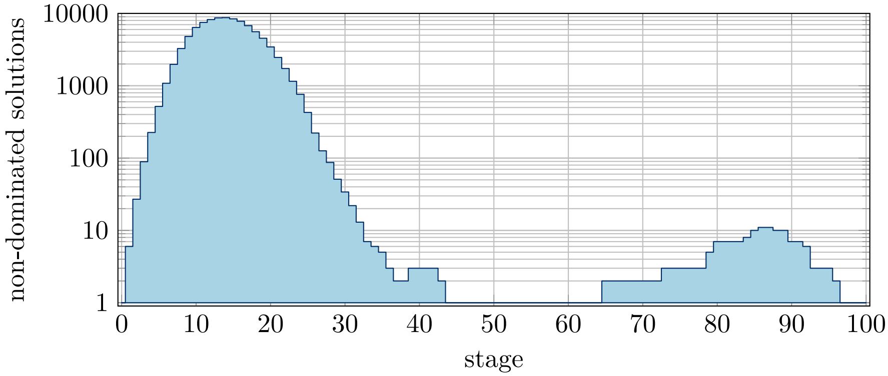


Figure 5.1: *Number of non-dominated solutions per stage for instance* ft10

5.2.3 Finding solutions

Trying to find the best solutions solely with DP we started with very small explorations within the DP state space by setting $H = 10$ and start with no bound in the first iteration. In the subsequent iterations we used the result of the previous iteration as an upper-bound. We iterated this until we did not improve the solution. It is profitable to rerun with a better upper bound and the same number of solutions to expand H, this is in contrast with the bounding used for the VRP described in section 4.1. The parallel head–tail adjustments described in section 5.1.1 can provide a higher lower bound when a lower upper bound is used. This affects the state space by bounding extra partial solutions. Besides, since the lower bound, which is used as estimated completion, changes the partial solutions selected to be expanded can change. This is an advantage over the dynamic bounding we developed for the CVRP in section 4.1, since there the value of the upper bound did not affect the estimation of the completion. When this iterative procedure did not improve the solution we repeated the process by setting $H = 100$ and using the best bounds from the runs with $H = 10$.

With these very limited state space we are able to find the optimal solution for 29 of the instances, some of which are quite large. However, for some small instances we cannot find the optimal solution with this very limited state space size, although we can find this solution when we use the optimal value as an upper bound as we have seen above. With $H = 100$ the running time on the large instances becomes quite high. Possibly, a better strategy would be to find a reasonable lower bound (see section 5.2.4), to use this lower bound as an upper bound for the DP algorithm using a very small value for bound H. When no solution is found, the DP algorithm can be repeated while the upper bound given to the algorithm could be slowly increased, until a solution is found.

Detailed results of these runs can be found in table A.2 on pages 130–139 in appendix A. Figure 5.2 shows the gap with the best known value of all the instances grouped by number of operations first and sorted by gap. It also shows the running time of the complete iterative process to achieve this gap.

5.2.4 Finding lower bounds

As some instances are not yet solved we also tried to improve known lower bounds. We have already discussed how to use DP to find optimal solutions and also as a heuristic for upper bounds, now we focus on tightening lower bounds. In order to prove that a value v is a valid lower bound we can run the DP algorithm for the JSSP using $v - 1$ as an upper bound. When the DP algorithm is run without any limitation on the state space size, such as bounding the number of expanded solutions by width H or bounding the number of expansions by E, it would find an optimal solution given $v - 1$ is a valid upper bound. If no solution is found we may conclude, since our instances take only integral values, that v is a valid lower bound on any solution.

When $v - 1$ is a valid upper bound the optimal solution would be found which would cost a lot of calculation time for the larger instances. To limit this

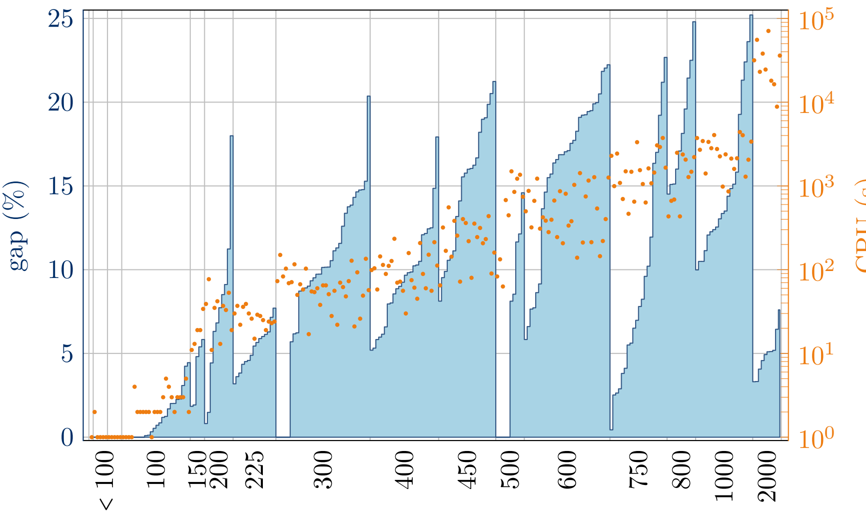


Figure 5.2: *The gap with the best known solution and running time for the 242 instances*

calculation time we set a stage width of $H = 10^5$ and let a run terminate as soon as this width would leave solutions unexpanded.

For the instances with known optimal values we tried to validate that the optimal value is a lower bound thereby proving the optimality of the known solution. As we can see in table A.3 (pages 140–141) the optimality of 121 of the 145 instances with known optimal values could be proven. Information on the 24 instances for which we could not prove the optimality is given in table A.4 (page 141).

For 16 of the 97 unsolved instances we could find a better lower bound than the current best known lower bound. For these instances we first increased the value of the current best known lower bound twice before using binary search to find the best lower bound we could prove using our DP algorithm with a maximal stage width of $H = 10^5$. The results and the new lower bounds can be found in table 5.5.

With this strategy it is typically the case that the DP algorithm takes much longer to run when the fact that a certain value is not a valid upper bound cannot be proven compared to the situation when this fact can be proven. This difference originates from the following consideration: if the fact that a value is not a valid upper bound cannot be proven, then the state space is filled to at least H partial solutions in the last stage before termination, while a value can only be proven not to be a valid upper bound if the number of solutions is less than H in each stage (i.e., H did not affect the size of the state space). Often when a bound can be proven to be a valid upper bound it is the case that early in the state space all solutions are bounded so no solution can be expanded and the algorithm stops. A good example of this difference can be found in the results for the instances *dmu12* and *dmu19*, both ran for 9 iterations. For the instance *dmu12* most of the times the algorithm could prove the upper bound to be invalid, leading to an increase of 63 in the lower bound with only a difference of 14 with the best known upper bound. However, for the instance *dmu19* the lower bound could only be increased by 3 leaving a gap of 96 with the best known upper bound. When we take a look at the run times for both instances we see that a large improvement for *dmu12* is done in less than 10% of the computation time required to achieve a small improvement for instance *dmu19*.

Instance	# J[a]	# M[b]	LB	UB	New LB	# I[c]	CPU[d]	Mem[e]
dmu06	20	20	2998	3244	*3042*	11	13110	858
dmu07	20	20	2815	3046	*2828*	11	10651	924
dmu12	30	15	3418	3495	*3481*	9	1479	869
dmu19	30	20	3669	3768	*3672*	9	16657	1536
dmu42	20	15	3172	3390	*3224*	11	13134	626
dmu44	20	15	3283	3488	*3299*	11	11615	777
dmu45	20	15	3001	3272	*3039*	11	7879	684
dmu51	30	15	3917	4167	*3954*	11	23102	1376
dmu52	30	15	4065	4311	*4094*	11	18488	943
dmu55	30	15	4140	4271	*4146*	8	34473	1025
dmu59	30	20	4217	4624	*4219*	3	5693	1349
dmu62	40	15	5033	5265	*5041*	11	33251	2163
dmu65	40	15	5105	5190	*5107*	3	15678	2026
dmu66	40	20	5391	5717	*5397*	12	17802	2467
swv07	20	15	1447	1594	*1457*	11	9539	788
ta50	30	20	1807	1923	*1808*	2	1233	1327

[a] # Jobs [c] # Iterations [d] Sum of CPU over all iterations (s)
[b] # Machines [e] Max Memory used over all iterations (MB)

Table 5.5: *Improvements of lower bounds found by DP*

5.3 All solutions for the JSSP

With the iterative DP algorithm described in section 3.2 not all optimal solutions will necessarily be found when using the indirect bounding in the DP algorithm described in section 2.3. According to this description we would add Ω to ϕ in the state space definition $\xi_{S \wr \vec{\alpha} \wr \vec{\eta}}$, resulting in $\xi_{S,\Omega \wr \vec{\alpha} \wr \vec{\eta}}$. As described briefly at the end of section 3.2, the bookkeeping variables $\beta(=\vec{\eta})$ allow for indirect domination of a partial solution ς that has an optimal completion $\mathring{\varsigma}$ by a partial solution ς' which has no optimal completion. This indirect domination results from the fact that the optimal completion $\mathring{\varsigma}$ of ς can be used to optimally complete ς' to solution $\mathring{\varsigma}'$. However, this completion $\mathring{\varsigma}'$ can be unordered, so the ordered sequence of $\mathring{\varsigma}'$ is not a completion of ς'. In other words, in the ordered sequence of $\mathring{\varsigma}'$ at least one operation of the completion has to be before the last operation in ς'.

In order to ensure that optimal solutions, of which partial solutions are dominated indirectly by a partial solution without a direct optimal completion, are found we have to alter the DP state space a little further. First, we have to find a way to identify partial solutions like ς' that have no optimal completion, but that can dominate partial solutions with an optimal completion. When partial solution ς' itself does not have an optimal completion but dominates another solution ς which has an optimal completion $\mathring{\varsigma}$, there is an operation o which will occur before the last operation of ς' in the ordered optimal solution $\mathring{\varsigma}'$ that results from adding the optimal completion of ς to ς'.

Without loss of generality, we assume that ς' consists of the sequence $\langle \varsigma'_a, o', \varsigma'_b \rangle$ and the optimal solution $\mathring{\varsigma}'$ starts with the sequence $\langle \varsigma'_a, o \rangle$ where o is an operation of the optimal completion. Now the partial solution ς'_a has an optimal completion ($\mathring{\varsigma}'$), the partial solution $\langle \varsigma'_a, o' \rangle$ however does not have an optimal completion. When the optimal solution $\mathring{\varsigma}'$ is found, Ω will not be empty for ς'_a. However, Ω will be empty for solution $\langle \varsigma'_a, o' \rangle$. Note that $\eta(\varsigma'_a, o) = 1$, since it is a feasible expansion, and that $\eta(\langle \varsigma'_a, o' \rangle, o) = 0$ as o is moved before o' in the ordered solution $\mathring{\varsigma}'$. The possibility to move o before o' while ς'_a is optimal is exactly what adds the possibility to the expansions of $\langle \varsigma'_a, o' \rangle$ to dominate partial solutions with an optimal completion. In fact $\langle \varsigma'_a, o' \rangle$ would have an optimal completion if we would not require the operations in the solutions to be ordered (according to proposition 2.2).

To prevent extensions of $\langle \varsigma'_a, o' \rangle$ to dominate other partial solutions we want to add an identifier to Ω. We cannot use a number n as this would indicate that partial solution $\langle \varsigma'_a, o' \rangle$ could be completed to the n-th found optimal solution. Also we want to identify the point in the sequence where this partial solution did not have an optimal completion anymore. With any of the (numeric) identifiers in Ω for ς'_a, and the last operation o_a of ς'_a the location where we have no optimal completion anymore is uniquely defined. When we add o' to the identifier all extensions of $\langle \varsigma'_a, o' \rangle$ are defined by this identifier. o is needed in the identifier to deduce when this identifier should be removed from the state where an expansion belongs to. When any expansion schedules a new operation on machine $m(o)$, o cannot be moved before o' anymore so the identifier can be removed. Actually $m(o)$ is sufficient in the identifier to be able to deduce when to remove it from Ω.

This results in the identifier "$n|o_a|o'|m$". This identifies an expansion of the n-th optimal solution up to operation o_a, which is expanded to o' and for which there is an operation that is to be scheduled on machine m for which an ordered expansion is no longer possible. For any partial solution $\varsigma'_a \mathbin{\diamond\!\!\Rightarrow} o'$ which has no optimal completion (yet), which is a expansion of a partial solution ς'_a with an optimal completion, and for which there is an operation $o (\neq o')$ which is an ordered expansion of ς'_a but not of $\varsigma'_a \mathbin{\diamond\!\!\Rightarrow} o'$, the partial solution $\varsigma'_a \mathbin{\diamond\!\!\Rightarrow} o'$ should be added to a state with identifier "$n|o_a|o'|m(o)$" added to Ω. Here, an optimal completion of ς'_a is the n-th optimal solution. When there are multiple operations that are an ordered expansion of ς'_a but not of $\varsigma'_a \diamond\!\!\Rightarrow o'$ multiple identifiers could be added to Ω. A single identifier for each of the machines such operation should be scheduled on. As soon as any operation is scheduled on machine m any identifier "$.|.|.|m$" should be removed from Ω.

This results in the separation of any extension of $\langle \varsigma'_a, o' \rangle$ from the state space with respect to dominance until with any expansion a new operation is scheduled on machine m. The effect of this separation is that these extensions will only dominate among themselves and will not dominate any other partial solution with an optimal completion. When $\varsigma'_a \diamond\!\!\Rightarrow o'$ indeed does not have an optimal completion, this domination is profitable as fewer partial solutions are considered. When $\varsigma'_a \diamond\!\!\Rightarrow o'$ does have an optimal completion (which is not found yet), the result is that possibly this solution is found in a earlier iteration of the iterative DP approach, since less domination for this solution takes place. All domination that occurs after the addition of the new identifiers would also have occurred without these new identifiers. The new identifiers only assure that the indirect domination between the partial solutions ς and ς' does not take place as soon as the optimal solution produced by the ordered sequence of the extension of ς' with the optimal completion of ς is found.

5.3.1 Finding all optimal solutions

For all instances with at most 10 jobs we tried to find all optimal solutions. Using the algorithm described in sections 3.2 and 5.3 we could find all optimal solutions for 19 of the 24 instances. For 5 instances the process ran out of the given 13 GB of memory, for one instance, *la05*, this happened in the first run. For this instance a single run is repeated without limiting the memory.

The implementation used for these results keeps the complete state space of the DP algorithm in memory. This state space is updated after each run to isolate the partial solutions of newly found optimal solutions as described in section 3.2. Furthermore, expansions of these partial solutions are possibly isolated as described in section 5.3. Also, two or more partial solutions which are equal with respect to the dominating criteria are all kept instead of only one.

The result of this implementation is that much more memory is used, since all stages are kept together in memory at the same time. The run time is shortened as large parts of the state space do not have to be re-expanded every iteration.

As we can see in table 5.6, the number of optimal solutions can vary a lot. We could not find a relation between the number of optimal solutions and the difficulty to find an optimal solution. For example, compare the results for *orb03* and *orb10* in tables 5.4 and 5.6.

For the instances which used all memory not all optimal solutions are found, table 5.7 gives the number of unique optimal solutions we found. These solutions were found in earlier runs of our algorithm than the run reported in section 5.3. Earlier implementations of our algorithm did find more solutions using the same amount of memory but did not have the guarantee to find all optimal solutions.

To give an impression of different optimal solutions all different schedules of *orb06* are shown in figure 5.3. The differences are marked in the schedules, these differences can be described as different orders for operations:

- o_{29} and o_{45} on m_5 (columns 1,2 vs. 3,4)
- o_{75} and o_{87} on m_9 together with o_{85} and o_{97} on m_{10} (columns 1,3 vs. 2,4)
- o_{61}, o_{72}, o_{89} and o_{95} on m_7 (rows 1 vs. 2 vs. 3–8)
- o_{81}, o_{94}, o_{98} and o_{99} on m_{10} (per row, where rows 1–3 are equal)

Instance	# J[a]	# M[b]	# Solutions	# I[c]	CPU[d]	Mem[e]
abz5	10	10	480	23	371	1420
abz6	10	10	2159	46	14	97
ft06	6	6	53	9	0	16
ft10	10	10	13120	36	248	822
la01	10	5	86173[f]	45	3983	13312
la02	10	5	66989	566	1194	732
la03	10	5	720	29	1	29
la04	10	5	83284	1243	3446	936
la05	10	5	682[f]	2	1686	15245
la16	10	10	47880[f]	109	27387	13312
la17	10	10	266573[f]	1304	1846522	13313
la18	10	10	42158	551	1480	1964
la19	10	10	960	34	43	222
la20	10	10	14016	222	162	466
orb01	10	10	9120	95	100	369
orb02	10	10	504	19	168	781
orb03	10	10	248	26	334	1214
orb04	10	10	96	19	44	157
orb05	10	10	288	17	81	319
orb06	10	10	32	6	79	300
orb07	10	10	162	10	25	115
orb08	10	10	302727[f]	107	118732	13398
orb09	10	10	108270	868	7445	3701
orb10	10	10	15951	158	74	328

[a] # Jobs
[b] # Machines
[c] # Iterations
[d] Sum CPU over all iterations (s)
[e] Max Memory used in all iterations (MB)
[f] Out of memory: Not all optimal solutions

Table 5.6: *All optimal solutions for small JSSP instances*

Figure 5.3: *Schedules for all optimal solutions of orb06*

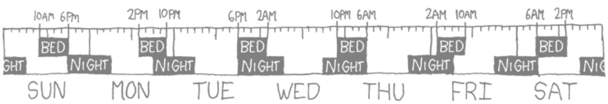
Instance	# Solutions in table 5.6	# Different Solutions found
la01	86173	278505
la05	682	692
la16	47880	189632
la17	266573	833178
orb08	302727	373020

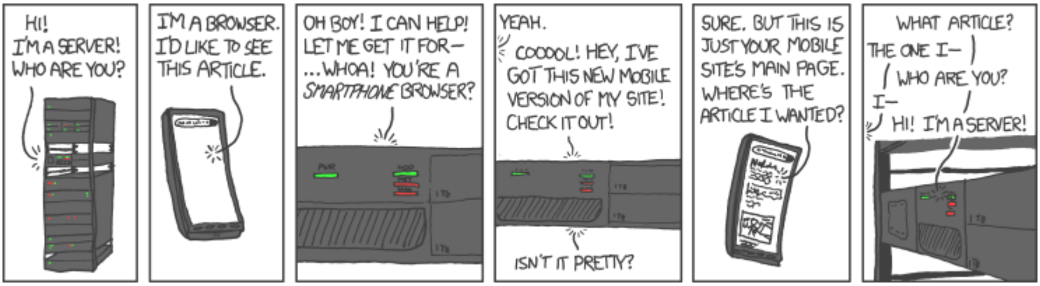
Table 5.7: *Number of found optimal solutions for small JSSP instances where not all are found*

For the instance *orb07* the algorithm is altered a bit as it contains an operation of zero length. Not all optimal solutions were initially found due to the definition of an ordered sequence. There exists an operation with a zero duration (operation 100) on a machine with a lower machine number as the preceding operation of the same job. If this operation is to be scheduled with the same finish time as its predecessor of the job there exist no feasible ordering to represent this. Since both operations have the same finish time, they should be ordered according to the index of the machine, however, this conflicts with the precedence relation given by the job. For our implementation this was solved by allowing this operation to follow the preceding operation of its job directly breaking the ordering described in this thesis.

Remark We defined the processing as $p_{mj} \in \mathbb{N}$. If this has to be generally extended to $p_{mj} \in \mathbb{N}_0$ proposition 2.2 does not provide a unique sequence. It is even possible that no sequence for a schedule exist. When $p_{mj} = 0$, we here assume the operation still has to visit machine m like any operation, however, the processing time can be neglected. When zero duration operations have to be taken into account a solution to fix this ordering could be to separate the operations with zero processing time within the ordering. That is for operations with the same end time any operation with non-zero processing time should precede any operation with zero processing time. After this the machine index can be used as tie breaker again. This way a schedule can have only multiple ordered sequences when multiple zero processing time operations are scheduled at the same time on the same machine, however, these can essentially be seen as different solutions.

All optimal solutions found for these and other instances are published on my web site with JSSP instances. Also the bounds described in appendix B can be found on this site. This web site can be found at `http://jobshop.jjvh.nl` [67].





SIX

The Job Shop Scheduling Problem with Scheduled Maintenances

In this chapter we extend the JSSP by adding extra constraints to the problem to incorporate maintenance of the machines into the problem. For this new problem we create a Mixed-Integer Programming model as well as an extension of the DP algorithm for the JSSP to incorporate these maintenances. We also extend the bounding described in section 5.1 to incorporate maintenances. To be able to do computational experiments we propose a method to create new instances with various characteristics. This chapter ends with a computational comparison of the DP algorithm for the Job Shop Scheduling Problem with scheduled Maintenances with the Mixed-Integer Programming model.

6.1 Adding maintenances

In practice the machines that are modeled in the JSSP may require maintenance in order to prevent them from breaking down. The repair cost and cost of the production lost during such a breakdown is in general much more expensive than scheduling some maintenance in advance. We assume that the maximum operating time without maintenance (uptime) U_i and the duration of the maintenance (downtime) D_i are deterministic and known in advance for each machine m_i. This problem can be seen as an extension of the single-machine problem studied in Qi, Chen, and Tu [100].

This problem we call the Job Shop Scheduling Problem with scheduled Maintenances (JSSPM) and is, to our best knowledge, not yet studied in literature. We show first that a solution cannot easily be constructed from an optimal JSSP Solution. Then we propose a Mixed-Integer Programming (MIP) formulation for the problem, working to our final goal for this section: to incorporate this maintenance in the DP algorithm for the JSSP.

A straightforward way of scheduling these maintenances would be to take the schedule of the optimal JSSP solution and schedule maintenances whenever the next operation on a machine would exceed the maximum uptime for that machine schedule a maintenance before that operation. However, this procedure is not necessarily optimal as we can demonstrate with the instance of the JSSP described by table 6.1.

	Job 1			Job 2			Job 3			Job 4	
	$m(o)$	$p(o)$		$m(o)$	$p(o)$		$m(o)$	$p(o)$		$m(o)$	$p(o)$
o_1	1	3	o_2	2	7	o_3	3	3	o_4	1	5
o_5	2	7	o_6	1	10	o_7	2	3	o_8	3	5
o_9	3	9	o_{10}	3	2	o_{11}	1	3	o_{12}	2	7

Table 6.1: *Instance of the Job Shop Scheduling Problem*

The optimal makespan for the JSSP is 25 with the schedule given in figure 6.1. When the maximum uptimes (U_i) are 10, 10 and 11 and the downtimes (D_i) are 2, 2 and 8 for machines m_1, m_2 and m_3, respectively, adding the necessary maintenances (R) in the optimal solution described above would produce the schedule shown in figure 6.2, with a value of 32. However, the optimal solution of the problem with maintenances has a value of 29. This solution is the one depicted in figure 6.3. Removing the maintenances from the optimal solution of an instance with maintenances does not lead necessarily to an optimal solution to the original JSSP instance as is shown in figure 6.4. The resulting schedule has a value of 26 instead of 25.

6

We conclude that we cannot easily create the optimal solution of the JSSP with maintenances from the optimal JSSP solution or vise-versa. The Job Shop Scheduling Problem with scheduled Maintenances is an NP-hard problem because it has as a special case the Job Shop Scheduling Problem. This case can be created by setting the maximum time that a machine can operate without maintenance to a value greater or equal than the sum of the processing times of all jobs in that machine. Another option is setting all downtimes to 0. However, it is possible that a JSSPM instance has no feasible solution while the JSSP has a feasible solution. This is the case when the time a machine can go without maintenance is shorter than one of the operations of that machine.

6.2 A Mixed-Integer Programming formulation

The Job Shop Scheduling Problem with scheduled Maintenances can be formulated in Mixed-Integer Programming extending the formulation presented by Applegate and Cook [7]. Recall the following notation:

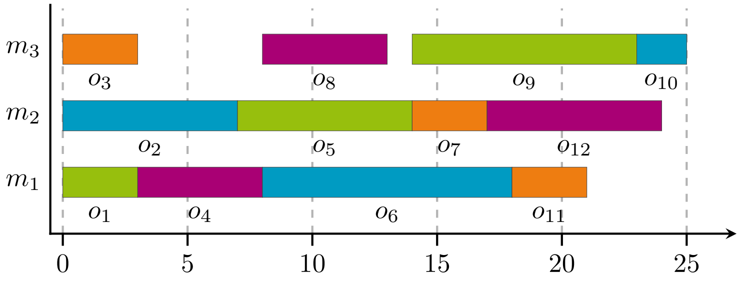


Figure 6.1: *Optimal solution of the JSSP instance of table 6.1*

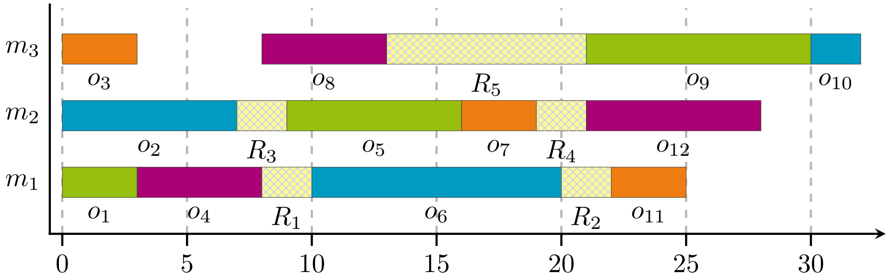


Figure 6.2: *Maintenances added into the schedule of figure 6.1*

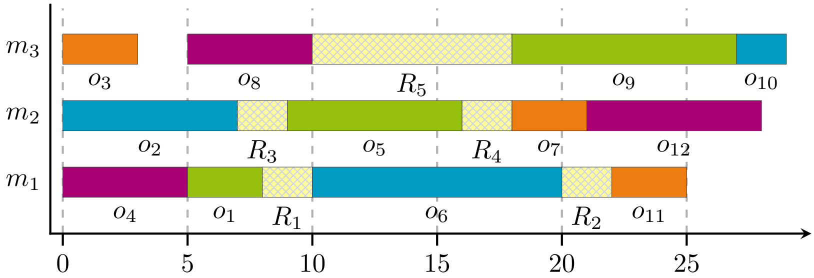


Figure 6.3: *Optimal solution of the JSSP instance of table 6.1 with maintenances*

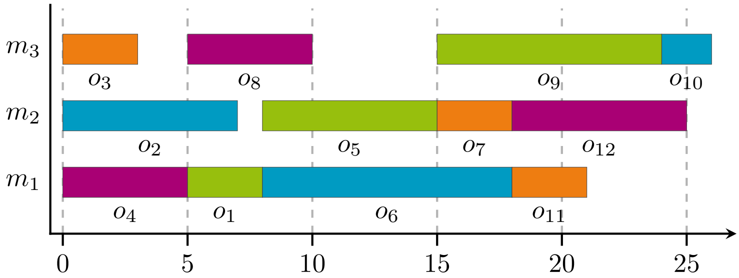


Figure 6.4: *The schedule of figure 6.3 with maintenances removed*

M	Number of machines
N	Number of jobs
$\mathcal{M} = \{m_1, \ldots, m_M\}$	Set of machines
$\mathcal{J} = \{j_1, \ldots, j_N\}$	Set of jobs
p_{ij}	Processing time of job j_j on machine m_i $(i = 1, \ldots, M$, $j = 1, \ldots, N)$
U_i	Maximum time that machine m_i can work without a maintenance $(i = 1, \ldots, M)$
D_i	Maintenance time for machine m_i $(i = 1, \ldots, M)$
$(\pi_j(1), \ldots, \pi_j(M))$	The sequence by which job j should visit the machines $(j = 1, \ldots, N)$

Each maximal set of operations scheduled on a single machine without any maintenance in between we call a group. Since we have N operations on each machine, we have at most N groups and at most $N-1$ maintenances on each machine. Let a group be indexed by $k \in \mathcal{G} = \{1, \ldots, N\}$.

For the MIP consider the following decision variables:

Z	Makespan.
x_{ij}	Starting time for processing of job j in machine i.
$u_{kij} = \begin{cases} 1 \\ 0 \end{cases}$	if job j is included in the k-th group processed on machine i, otherwise.
$y_{kijl} = \begin{cases} 1 \\ 0 \end{cases}$	if job j is processed after job l in the k-th group of machine i, otherwise.
w_{ki}	Starting time for the k-th maintenance in machine i.

6

With these variables a MIP formulation for the JSSPM is the following:

$$\min \quad Z \tag{6.1 a}$$

$$\text{s.t.} \quad x_{\pi_j(i)j} \geq x_{\pi_j(i-1)j} + p_{\pi_j(i-1)j}, \qquad i \in \mathcal{M} \setminus \{m_1\};\ j \in \mathcal{J} \tag{6.1 b}$$

$$Z \geq x_{\pi_j(M)j} + p_{\pi_j(M)j}, \qquad j \in \mathcal{J} \tag{6.1 c}$$

$$\sum_{j \in \mathcal{J}} u_{kij} p_{ij} \leq U_i, \qquad i \in \mathcal{M};\ k \in \mathcal{G} \tag{6.1 d}$$

$$x_{ij} \geq x_{il} + p_{il} - (1 - y_{kijl})\, I, \qquad i \in \mathcal{M};\ j, l \in \mathcal{J};\ k \in \mathcal{G} \tag{6.1 e}$$

$$w_{ki} \geq x_{ij} + p_{ij} - (1 - u_{kij})\, I, \qquad i \in \mathcal{M};\ j \in \mathcal{J};\ k \in \mathcal{G} \setminus \{N\} \tag{6.1 f}$$

$$x_{ij} \geq w_{ki} + D_i - \left(1 - u_{(k+1)ij}\right) I, \qquad i \in \mathcal{M};\ j \in \mathcal{J};\ k \in \mathcal{G} \setminus \{N\} \tag{6.1 g}$$

$$u_{(k+1)ij} \leq \sum_{l \in \mathcal{J}} u_{kil}, \qquad i \in \mathcal{M};\ j \in \mathcal{J};\ k \in \mathcal{G} \setminus \{N\} \tag{6.1 h}$$

$$u_{kij} + u_{kil} \leq 1 + y_{kijl} + y_{kilj}, \qquad i \in \mathcal{M};\ j, l \in \mathcal{J};\ k \in \mathcal{G} \tag{6.1 i}$$

$$\sum_{k \in \mathcal{G}} u_{kij} = 1, \qquad i \in \mathcal{M};\ j \in \mathcal{J} \tag{6.1 j}$$

$$x_{ij} \geq 0, \qquad i \in \mathcal{M};\ j \in \mathcal{J} \tag{6.1 k}$$
$$u_{kij} \in \{0,1\}, \qquad i \in \mathcal{M};\ j \in \mathcal{J};\ k \in \mathcal{G} \tag{6.1 l}$$
$$y_{kijl} \in \{0,1\}, \qquad i \in \mathcal{M};\ j,l \in \mathcal{J};\ k \in \mathcal{G} \tag{6.1 m}$$
$$w_{ki} \geq 0, \qquad i \in \mathcal{M};\ k \in \mathcal{G} \setminus \{N\} \tag{6.1 n}$$

Constraints (b) assure that all operations, except the first operation, of a job starts only after the previous operation of that job (on another machine) is finished. Constraints (c) guarantee that the makespan is larger than, in fact it will be equal to, the finish time of the last operation of the last job to be completed. Constraints (d) limit the active time for each group on each machine to the maximum uptime of the machine, so the maximum time a machine can operate without maintenance is satisfied. In constraints (e–g) I denotes an arbitrary large number. A save value for I would be a value larger than the sum of the processing times of all operations and maintenances. Constraints (e) assure that when job j is processed after job l on the same machine i and the same group k, job j starts after job l is finished. Constraints (f) assure that the k-th maintenance of each machine is scheduled after all jobs of group k are finished on that machine, while constraints (g) assure that all jobs of group $k+1$ on a machine start after the k-th maintenance is finished. Constraints (h) ensures that if the k-th group for machine i is empty all following groups are also empty. Constraints (i) guarantee that variable y_{kijl} or y_{kilj} is equal to 1 if in fact job j and job l are both processed in the k-th group on machine i. Constraints (j) guarantee that the operation of job j on machine i belongs to exactly one group. Finally, constraints (k–n) are domain constraints.

6.3 Dynamic Programming

6

To incorporate the maintenances into the DP algorithm for the JSSP we first have to define the maintenances in a similar way as the operations. Furthermore, we have to define an ordered sequence of operations and maintenances. Finally, we have to change the state definition to keep track of the current uptime of each machine to ensure the principle of optimality.

Let us define the maintenances (or repairs) similar to the set of operations as

$$\mathcal{R} = \left\{ R_1, \ldots, R_{M\times(N-1)} \right\},$$

where each maintenance R_i is performed of machine m_j with $j = i \pmod{M}$ as the $\lceil \frac{i}{M} \rceil$-th maintenance on that machine. This results in a fixed ordering of the maintenances $R_i, R_{i+M}, \ldots, R_{i+M\times(N-2)}$ for each machine m_i. Extend the definitions of $m(\cdot)$ and $p(\cdot)$ for maintenances by defining $m(R)$ as the machine performing maintenance R and $p(R) = D_{m(R)}$ as the time needed for the maintenance. Let $\mathcal{T} = \mathcal{O} \cup \mathcal{R}$ be the set of *tasks*, i.e., operations and maintenances, to be scheduled for an instance of the JSSPM.

Now we define a sequence of tasks similarly to the sequences of operations for the JSSP, which is ordered when the tasks are ordered according to the completion time of each task in a no-idle schedule. Note that in a no-idle schedule each maintenance will directly follow a task on its machine. This follows directly from the fact that any task that has to be scheduled before it according to a precedence relation is always performed on the same machine. Such an ordered sequence of a schedule representing a complete solution for the JSSPM does not necessarily contain all possible maintenances $\mathcal{R}$, as this represents the maximal set of maintenances.

To solve the JSSPM by DP we perform a DP algorithm over the set of all tasks $\mathcal{T}$. Before we look at the state definition we extend the definitions of $\varepsilon(S)$, $\eta(\varsigma_S)$ and $\alpha(\varsigma_S, o)$ to include $\mathcal{R}$ as elements of S and possible next tasks t in the case of α (i.e., $\alpha(\varsigma_S, t)$). As we will see later the definitions of η and α, will have to be changed slightly further. For each machine there is at most one maintenance available to be scheduled as the order of the maintenances is fixed. So, $|\varepsilon(S)| \leq N + M$, one operation for each job and one maintenance for each machine. Since the definition of $\varepsilon(S)$ is changed, the length of $\vec{\eta}$ and $\vec{\alpha}$ is also increased.

Recall the state definition $\xi_{S,\vec{\eta}\wr\vec{\alpha}}$ of proposition 2.7, where still $\vec{\eta} \subset \phi$. Similarly to the DP algorithm of the JSSP we first create an optimal DP algorithm without bookkeeping variables and then move $\vec{\eta}$ to the bookkeeping variables β. The fact that each job visits each machine exactly once in the general JSSP is never used in the definitions leading up to proposition 2.7 as well as in its proof. Actually the DP algorithm for the JSSP also works for instances where each job can visit each machine an arbitrary number of times, the different jobs also do not need to have an equal number of operations. Accordingly, with this state definition all tasks can be optimally scheduled as if it were a regular JSSP instance with M extra jobs of which all operations have to take place on the same machine.

Naturally, the feasibility regarding the maximum uptime U is lost. For two solutions $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}}$ and $\varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ in the same state $\xi_{S,\vec{\eta}}$ where $\varsigma_{S,\vec{\eta}\wr\vec{\alpha}} \geq \varsigma'_{S,\vec{\eta}\wr\vec{\alpha}'}$ ($\vec{\alpha} \geq \vec{\alpha}'$), all feasible expansions and completions of ς' must be dominated by the same expansion made to ς. However, it is possible that the uptime on a machine is longer for a completion of ς than it is for the same original completion of ς'.

To be able to test the feasibility regarding the maximum uptime U we introduce variables u_i for each machine m_i which represents the maximal processing time left until a maintenance is required, so for $S = \emptyset$ we have $u_i = U_i$. When we combine the variables u_i into an array $\vec{u}$ and define $\geq$ for $\vec{u}$ similar to $\vec{\alpha}$ and γ using $\geq$ for each element-wise compare. So we have $\vec{u} \geq \vec{u}'$ when all elements in $\vec{u}$ are greater or equal to their corresponding elements in $\vec{u}'$. When we add $\vec{u}$ to γ ($\gamma = \{\vec{\alpha}, \vec{u}\}$) we get state definition $\xi_{S,\vec{\eta}\wr\vec{\alpha},\vec{u}}$ and we restore the optimality principle. For two solutions $\varsigma_{S,\vec{\eta}\wr\vec{\alpha},\vec{u}}$ and $\varsigma'_{S,\vec{\eta}\wr\vec{\alpha}',\vec{u}'}$ in the same state $\xi_{S,\vec{\eta}}$ where $\varsigma_{S,\vec{\eta}\wr\vec{\alpha},\vec{u}} \geq \varsigma'_{S,\vec{\eta}\wr\vec{\alpha}',\vec{u}'}$ ($\vec{\alpha} \geq \vec{\alpha}'$ and $\vec{u} \geq \vec{u}'$), all feasible expansions and completions of ς' are dominated by the same expansion made to ς. The sum of the uptimes until the next maintenance for each machine in this completion is smaller or equal to the value in $\vec{u}'$, corresponding to the machine, otherwise it would not be a feasible completion of ς'. Since the value in $\vec{u}$ corresponding to the same machine is as least as high, the maximum uptime is not violated when completing ς with this completion.

However, we still need a final alteration to the algorithm to be able to find optimal solutions for the JSSPM. With the current state definition the solution with the earliest completion time is found for a schedule with exactly $N-1$ maintenances per machine. To find the optimal solution for the JSSPM we simply have to use $p(R)=0$ when calculating $\eta(\varsigma_S)$ or $\alpha(\varsigma_S,R)$ creating an expansion $\varsigma_{S,\vec{\eta}\wr\vec{\alpha},\vec{u}} \diamond\!\!\Rightarrow R$. Now still all $M\times(N-1)$ maintenances are added to the schedule, only the maintenances performed after all operations on a machine are finished have zero time. Note that using $p(R)=0$ only depends on the state variable S.

To improve the running time of the algorithm we can skip the expansion of all maintenances after all operations on a single machine are performed. When these expansions are not made, the state space will not be completely filled, since all solutions in states $\xi_{S,\vec{\eta}\wr\vec{\alpha},\vec{u}}$ with $S\supseteq\mathcal{O}$ will not have any feasible expansions. To find the feasible solution all solutions in such states have to be checked against the best found solution so far. Furthermore, the entry in $\vec{\alpha}$ and $\vec{u}$ corresponding to m_i can be removed when $\{\, o\in\mathcal{O}\setminus S \mid m(o)=i \,\}=\emptyset$, that is when all operations on m_i are scheduled. Now $\vec{\alpha}$ is automatically reduced to $C_{\max}$ when $S\supseteq\mathcal{O}$, as it is with the DP algorithm for the regular JSSP.

The feasibility test for any expansion $\varsigma_S \diamond\!\!\Rightarrow o$ regarding the uptime of a machine by testing $p(o)\geq u_{m(o)}$ can also be done beforehand by setting $\eta(\varsigma_S,o)=0$. To further improve the running time, the expansions $\varsigma_S \diamond\!\!\Rightarrow R$ with $u_i=U_i$ for $i=m(R)$ can pre prohibited, as adding the maintenance will not improve the allowed remaining processing times for operations. This feasibility check can also be performed on the previous expansion and we can set $\eta(\varsigma_S,R)=0$. Also we can render any solution infeasible where we have that $\eta(\varsigma_S,R)=0$ with $m(R)=i$ and for all $o\in\{\, o'\in\mathcal{O}\setminus S \mid m(o')=i \,\}$ we have that $p(o)>u_i$, since no operation will be able to be scheduled before a maintenance is scheduled and the maintenance cannot be scheduled in an ordered sequence anymore.

An upper bound on complexity of this DP algorithm can be found along the same lines as described in section 2.3. For the DP algorithm for the JSSP we started with —using B instead of U— $\mathcal{O}\big(B(B+N)MN2^{MN}\big)$, we could limit the number of expansions to N, we could reduce by the factor $\left(\frac{2^M}{M+1}\right)^N$ and we found $B=\mathcal{O}\Big(\frac{p_{\max}{}^N}{\sqrt{N}}\Big)$. For the JSSPM we have $MN+M(N-1)$ tasks and each solution can be expanded with at most $N+M$ tasks. So we start with $\mathcal{O}\big(B(B+N+M)(M+N)2^{M(2N-1)}\big)$. We have M extra precedence relations of length $N-1$ for the maintenances giving a reduction factor of $\left(\frac{2^{N-1}}{N}\right)^M$ next to the still existing $\left(\frac{2^M}{M+1}\right)^N$. This results in $\mathcal{O}\big(B(B+N+M)(M+N)N^M(M+1)^N\big)$.

For B we can create an antichain for the set of non-dominating different values of $\gamma=\{\vec{\alpha},\vec{u}\}$ similar to section 2.3. In γ we have N values of at most $p_{\max}$ and M values of at most $U_{\max}$, where $U_{\max}=\max_{m_i\in\mathcal{M}} U_i$. If we take

$T_{\max} = \max\{p_{\max}, U_{\max}\}$ we can, using the intermezzo in section 2.3, conclude that

$$B \leq p_{\max} \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \frac{(T_{\max}+1)^{N+M}}{\sqrt{\frac{1}{3}(N+M)({T_{\max}}^2 + 2T_{\max})}} = \mathcal{O}\left(\frac{{T_{\max}}^{N+M}}{\sqrt{N+M}}\right).$$

From this we obtain the following complexity for the DP algorithm for the JSSPM:

$$\mathcal{O}\left(\frac{{T_{\max}}^{N+M}}{\sqrt{N+M}}\left(\frac{{T_{\max}}^{N+M}}{\sqrt{N+M}} + N + M\right)(M+N)\,N^M(M+1)^N\right)$$
$$\mathcal{O}\left(\left(\frac{{T_{\max}}^{2(N+M)}}{N+M} + \frac{(N+M){T_{\max}}^{N+M}}{\sqrt{N+M}}\right)(M+N)\,N^M(M+1)^N\right)$$
$$\mathcal{O}\left(\left({T_{\max}}^{2(N+M)} + (N+M)\sqrt{N+M}{T_{\max}}^{N+M}\right)N^M(M+1)^N\right)$$
$$\mathcal{O}\left({T_{\max}}^{2(N+M)}N^M(M+1)^N\right).$$

6.4 Bounding for the JSSPM

To limit the size of the DP state space for the JSSPM effectively the bounding described in section 5.1.1, although it can be used, should be improved. In this section we describe the parallel head–tail adjustments of Brinkkötter and Brucker [21] with alterations to incorporate maintenances, following the lines of Brinkkötter and Brucker [20]

6

To each machine k we add $\mathcal{N}_k$ maintenance operations which are sequenced in a fixed order using precedence relations. These operations with their relations can be seen as an extra job only to be performed on that particular machine. Furthermore, we ensure that between each pair of consecutive maintenances on a machine at least one operation could possibly be planned.

Let I_k be the set of operations on machine k. Let $\mathcal{N}_k$ be a lower bound on the number of maintenances required to schedule all operations in I_k and let $\breve{I}_k$ be a set of $\mathcal{N}_k$ maintenances. Define $\bar{I}_k = I_k \cup \breve{I}_k$. See the intermezzo below for a possible way to determine $\mathcal{N}_k$.

Intermezzo: A lower bound for the one-dimensional bin packing problem

The minimal number of maintenances needed between a set of operations can be seen as a one-dimensional bin packing problem. The set of operations to be planned represent indivisible items, with size equal to their respective processing times, that are to be put in bins of size equal to the maximal uptime U. The goal is to minimize the number of bins.

Since each bin represents the maximal uptime, maintenances are needed to separate these bins, thus the minimum number of maintenances needed is equal to the number of bins needed minus 1.

In Martello and Toth [83, chap. 8.3] multiple lower bounds on the one-dimensional bin packing are given. As an example we briefly show two of these bounds.

Let I be the set of operations that have to be planned. The first, straightforward, lower bound L_1 can be found under the assumption that any item can be arbitrary split into two bins. By filling all bins fully we get

$$L_1 = \left\lceil \frac{\sum_{o \in I} p_o}{U} \right\rceil .$$

For the second lower bound L_2 we create three disjunct subsets J_1, J_2 and J_3 of I according to an integral parameter $\alpha \in [0, U/2]$.

$$\begin{aligned} J_1 &= \{o \in I \mid p_o > U - \alpha\} \\ J_2 &= \{o \in I \mid U - \alpha \geq p_o > U/2\} \\ J_3 &= \{o \in I \mid U/2 \geq p_o \geq \alpha\} \end{aligned}$$

Because sets J_1 and J_2 fill more than half a bin items from these sets cannot be combined. Items from set J_3 can only be combined with J_2 and J_3, if the total size of these items is larger than the remaining capacity of the bins that are filled by items of J_2 the logic of L_1 is applied to the remaining size. This results in a bound depending on α of

$$L_2(\alpha) = |J_1| + |J_2| + \max\left\{0, \left\lceil \frac{\sum_{o \in J_3} p_o - \left(|J_2| U - \sum_{o \in J_2} p_o\right)}{U} \right\rceil \right\} .$$

Finally, the second bound becomes

$$L_2 = \max\{L_2(\alpha) \mid 0 \leq \alpha \leq U/2\} .$$

Note that L_2 dominates L_1 as $L_2(0) = |J_2| + \max\{0, L_1 - |J_2|\}$.

First we take a look at the operations on a single machine. Let U_k and D_k be the maximum uptime and downtime on this machine, respectively.

For each operation $o \in \bar{I}_k$ we create a head r_o and tail q_o. The corresponding head-tail problem is to find a schedule such that each operation does not start before its head, that maximum uptime between maintenances of length D_k does not exceed U_k and for which $\max_{o \in I_k} \{C_o + q_o\}$ is minimal, where C_o is the finish time of operation o.

However, as we estimate the number of maintenances upfront we modify the problem slightly. We do not require that the maximal uptime between maintenances is D_k, as it is possible that there is no feasible solution with just

$\mathcal{N}_k$ maintenances since this is a lower bound. We just require that there is a normal operation between each maintenance as well as before the first and after the last maintenance. The corresponding head-tail problem is now to find a schedule such that each operation does not start before its head, before and after each maintenance of length D_k at least a normal operation is scheduled and for which $\max_{o \in I_k} \{C_o + q_o\}$ is minimal.

The preemptive variant of this problem can be solved by constructing a schedule from left to right by applying the same JPS scheduling rule as used in section 5.1.1:

> At time t schedule an available operation with the largest tail, until this operation is finished or the time defined by the smallest head with $r > t$.

We call a schedule constructed in this way a Jackson's preemptive schedule with scheduled Maintenances (JPSM). The only difference with the regular JPS is the inclusion of the minimal set of maintenances $\breve{I}_k$ in the JPSM.

Now we apply the same rules briefly described in section 5.1.1. Consider a partial schedule created following the rules for the JPSM at time $t = r_w$ given by the head of some operation $w \in \bar{I}_k$. Let again p_w^+ be the remaining processing time for operation $w \in \bar{I}_k$ at time t. Let $\mathcal{UB}$ be an upper bound on the optimal value of the non-preemptive head-tail problem. If for two operations $w, o \in \bar{I}_k$, $o \neq w$, we have that

$$r_w + p_w + p_o + q_o > \mathcal{UB}, \tag{6.2}$$

operation w cannot start before operation o. So $o \prec w$, and we may set r_w to:

$$r_w = \max\{r_w, r_o + p_o\}.$$

Another condition we can use to increase the value of r_w is the following. If we have a subset $Y_k \subset \bar{I}_k$ with $Y_k \subseteq \left\{o \in \bar{I}_k \mid p_o^+ > 0\right\} \setminus \{w\}$ the condition

$$r_w + p_w + \sum_{o \in Y_k} p_o^+ + \min_{o \in Y_k} q_o > \mathcal{UB} \tag{6.3}$$

holds, operation w cannot start before $r_w + 1$ in any optimal schedule. This can be seen as follows: Assume that we have an optimal schedule ψ in which operation w starts at r_w. Then using exchange arguments we can transform ψ into a schedule ψ' in which the operations of ψ' are performed on the same times as the JPSM until time r_w without increasing $\max_{o \in \bar{I}_k} \{C_o + q_o\}$. Since $\mathcal{UB}$ is an upper bound on the optimal value of $\max_{o \in \bar{I}_k} \{C_o + q_o\}$ for ψ' must hold that

$$r_w + p_w + \sum_{o \in Y_k} p_o^+ + \min_{o \in Y_k} q_o \leq \mathcal{UB}$$

which contradicts inequality (6.3). We can even set

$$r_w = r_w + \sum_{o \in Y_k} p_o^+.$$

This can be seen as follows: Schedule the remainder p_o^+ in unit steps of all operations o in Y_k disregarding their heads r_o following the rules for the JSSPM in order of decreasing tail values q_o. During this process inequality (6.3) continuously holds before scheduling any unit step. Also operation w cannot start before all operations in Y_k are finished, which may further delay the head r_w of operation w. Note that inequality (6.2) is implied by inequality (6.3) when $p_o = p_o^+$, so we consider inequality (6.2) after inequality (6.3) and only for operations where $p_o > p_o^+$.

To find the maximal set K_w^* satisfying inequality (6.3) we use the following procedure.

1. Let $K_w^+ = \left\{o \in \bar{I}_k \,\middle|\, p_o^+ > 0\right\} \setminus \{w\}$. Sort the operations in K_w^+ according to non-decreasing values of q_o.

2. With respect to this sorted sequence let $o_w \in K_w^+$ be the first operation satisfying
$$r_w + p_w + \sum_{\substack{o \in K_w^+ \\ q_o \geq q_{o_w}}} p_o^+ + q_{o_w} > \mathcal{UB}$$

3. Set
$$K_w^* = \left\{o \in K_w^+ \,\middle|\, q_o \geq q_{o_w}\right\}.$$

Now for K_w^* the condition in inequality (6.3) is clearly satisfied. Furthermore, if Y_k is a set satisfying inequality (6.3), let $q_{Y_k} = \min_{o \in Y_k} q_o$, then

$$Y_k = \left\{o \in Y_k \,\middle|\, q_o \geq q_{Y_k}\right\} \subseteq \left\{o \in K_w^+ \,\middle|\, q_o \geq q_{Y_k}\right\} \subseteq K_w^* = \left\{o \in K_w^+ \,\middle|\, q_o \geq q_{o_w}\right\},$$

due to the definition of o_w. So K_w^* is the maximal set satisfying inequality (6.3).

It is possible that the definitions of K^* define a cyclic relation, that is $o \in K_w^*$ and $w \in K_o^*$. In that case operations o and w should wait indefinitely for each other and the provided upper bound is not a valid upper bound.

Finally, regarding the maintenances we can increase the heads of the maintenances by using the following. Let $o, w \in \breve{I}_k$ and without loss of generality assume that $o \prec w$. We set $h_k^{\min} = r_o + p_o$ which is the first possible end of maintenance o, so $h_k^{\min}$ is a lower bound on the start of maintenance w. However, since there should be at least one operation $v \in I_k$ such that it fits between maintenances o and w. Let $\mathcal{F}_k$ be the set of normal operations that fit between these maintenances and can be scheduled after $h_k^{\min}$, i.e.,

$$\mathcal{F}_k = \left\{v \in I_k \,\middle|\, \max\left\{h_k^{\min}, r_v\right\} + p_v + q_v \leq \mathcal{UB}\right\}.$$

Now we can set $h_k^{\max}$ as the first time any operation scheduled after maintenance o can finish to

$$h_k^{\max} = \min_{v \in \mathcal{F}_k} \left\{\max\left\{h_k^{\min}, r_v\right\} + p_v\right\}.$$

So we can set the head of maintenance w to $r_w = \max\left\{r_w, h_k^{\max}\right\}$. To update the head for the first maintenance we can set $h_k^{\min} = 0$.

6.4.1 Updating heads and tails on a single machine

In this section we present an example of the procedure that dynamically updates the heads and tails on a single machine. We update the heads dynamically, i.e., we define $\tilde{r}_o$ as the temporary head which is updated during a single creation of a JPSM. First, $\tilde{r}_o$ is initialized with r_o, it is updated and fixed when $t = \tilde{r}_o$.

Consider the following example with a given upper bound of $\mathcal{UB} = 14$ and a maximum uptime $U = 3$ and a downtime $D = 2$.

Operation	p_o	r_o	q_o
o_1	2	2	1
o_2	2	3	4
o_3	2	3	8

Maintenance	p_o	r_o	q_o
R_1	2	0	0
R_2	2	0	0

It is easy to see that there are at least 2 maintenances needed, which are already given in the table as R_1 and R_2.

First, we construct the JPSM forward. At time $t = 0$, we have no operation available and also the time of the first maintenance cannot be determined as no head is fixed yet. At $t = 2$, o_1 becomes available we conclude that $K^*_{o_1} = \emptyset$ so we start with one time unit of operation o_1. Furthermore, as we have fixed $\tilde{r}_{o_1} = 2$ we set $h_k^{\max} = 5$. At $t = 3$, we conclude that $K^*_{o_2} = \{3\}$ and $K^*_{o_3} = \emptyset$, so $\tilde{r}_{o_3} = 3$, $o_3 \prec o_2$ and we schedule o_3. At $t = 5$, we conclude that $\tilde{r}_{R_1} = 5$ and $K^*_{o_2} = \emptyset$, so we set $h_k^{\max} = 9$ and schedule operation o_2. Finally at $t = 7$, $t = 8$ and $t = 10$, we schedule the remainder of o_1, R_1 and R_2, respectively. The resulting schedule of the JPSM and the schedules of the following iterations are given in figure 6.5. The new set of heads and tails are now

Operation	p_o	r_o	q_o
o_1	2	2	1
o_2	2	**5**	4
o_3	2	3	8

Maintenance	p_o	r_o	q_o
R_1	2	**5**	0
R_2	2	**9**	0

Second, we reverse the roles of the heads and tails and create the JPSM backwards. At time $t = 14$, again we have no operation available and also the time of the first maintenance cannot be determined, since no head is fixed yet. At time $t = 13$, we conclude that $K^*_{o_1} = \emptyset$, set $\tilde{q}_{o_1} = 1$ and $h_k^{\max} = 11$ and schedule

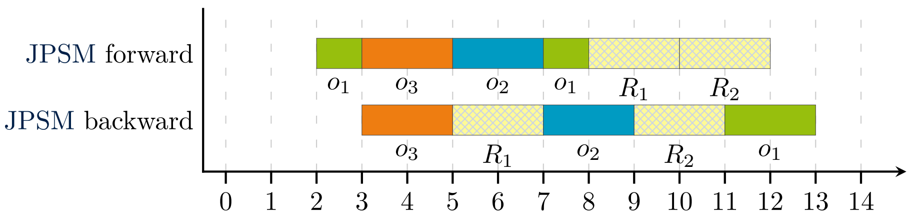


Figure 6.5: *JPSM forward and backward*

o_1. At time $t = 11$, we can set $\tilde{q}_{R_2} = 3$, $h_k^{\max} = 7$ and schedule R_2 for a single time unit. At time $t = 10$, we conclude that $K_{o_2}^* = \{R_2\}$ and schedule the remainder of R_2. At time $t = 9$, we set $\tilde{q}_{o_2} = 5$ and schedule o_2. At time $t = 7$, we set $\tilde{q}_{R_1} = 7$ and schedule one unit from R_1. At time $t = 6$, we can conclude using inequality (6.2) that $o_3 \prec R_1$, so we can set $\tilde{q}_{o_3} = 9$ and schedule the remainder of R_1. At time $t = 5$, we schedule o_3. In figure 6.5 the JSSPM backward is also shown. The new set of heads and tails are now

Operation	p_o	r_o	q_o
o_1	2	2	1
o_2	2	5	**5**
o_3	2	3	**9**

Maintenance	p_o	r_o	q_o
R_1	2	5	**7**
R_2	2	9	**3**

Finally, another forward calculation of JPSM will set $r_{o_1} = 11$ and $r_{o_2} = 7$ but the schedule does not change from the last backward iteration. Now all operations are fixed and further iterations give no further changes.

6.4.2 Updating heads and tails on all machines

Using the algorithms described in Brinkkötter and Brucker [20] we can update the heads on all machines simultaneously. In this section we describe how to update the heads and tails on all machines. The algorithms to update the heads are slightly modified versions of the ones in [20].

When we consider the JSSPM we have a set of operations $\bar{I}_k$ for each machine k $(k = 1, \ldots, M)$. We apply the JPSM and update the heads for each of these sets simultaneously. Note that we have precedence relations from the jobs between operations of different sets $\bar{I}_k$. If we increase the head of operation o and operation w is a successor of o $(o \prec w)$ we can set the head of operation w to $r_w = \max\{r_w, r_o + p_o\}$. Similarly we can update the tail of o when we update the tail of w.

Algorithm 6.1 gives a global overview of the global algorithm to update the heads and tails of all operations.

In the following functions a few new variables are used. Variable $\mathcal{LB}^k$ is the lower bound for the head-tail problem of the set of all operations $\bar{I}_k$ on machine k, it is defined as

$$\mathcal{LB}^k = \max_{o \in \bar{I}_k} \{C_o + q_o\}.$$

This finally gives $\mathcal{LB} = \max_{k=1}^{M} \mathcal{LB}^k$ as lower bound for the complete JSSPM. Furthermore, we have t_k and t_k^{req} as the current local time and the next relevant local time on machine k. We define $\tilde{r}_o$ as the head of operation o that is updated to distinguish it from the given head r_o before the heads are updated. Finally $\mathcal{M}^+$ is the set of indexes of all machines which have not yet finished all their operations.

The function **UpdateHeads** in algorithm 6.2 calculates the improved lower bound and updates the heads for all operations. This is done by applying the JPSM to all machines simultaneously and updating the heads accordingly.

Algorithm 6.1 Iteratively update heads and tails

Input: An upper bound $\mathcal{UB}$
Output: A new lower bound $\mathcal{LB}$

$\mathcal{LB} = 0$

$\mathcal{LB} = \max\{\mathcal{LB}, \textbf{UpdateHeads}(\mathcal{UB})\}$
Switch roles of heads and tails

repeat
  $\mathcal{LB} = \max\{\mathcal{LB}, \textbf{UpdateHeads}(\mathcal{UB})\}$
  Switch roles of heads and tails
until No heads and tails are updated

return $\mathcal{LB}$

First the data and the state for each machine are initialized by **InitData** (algorithm 6.6) and **UpdateState** (algorithm 6.7). Then, iteratively a single step for the JPSM is performed by **JPSMstep** (algorithm 6.3) on the machine with the lowest relevant local time, i.e., a machine m_k for with

$$t_k^{\text{req}} = \min_{k' \in \mathcal{M}^+} t_{k'}^{\text{req}}.$$

Finally in function **SetHeads** (algorithm 6.8) the heads of all normal operations are set to their newly obtained values.

The function **JPSMstep** in algorithm 6.3 carries out a single step of the JPSM on a machine m_k, updates the head relevant for this step including possibly heads of successors and it updates t_k and t_k^{req}. Before we examine **JPSMstep** in detail we need to introduce a bit more notation. To allow uniform formulation of the following functions we introduce a dummy operation a_k for each machine m_k with $p_{a_k} = \infty$, $r_{a_k} = 0$ and $q_{a_k} = -\infty$. This operation a_k will be processed on machine m_k when no other operation $o \in \bar{I}_k$ is available, i.e., operation a_k fills the idle time on machine m_k. $h_k^{\min}$ is the first possible finish time of the last maintenance made available on machine k and $h_k^{\max}$ is the first possible time any normal operation can finish that is started on or after $h_k^{\min}$. $h_k^{\max}$ is a lower bound on the head of the next maintenance to be made available on machine k. Let $\textsc{Indegree}_o$ be the number of direct predecessors of o in the normal operations $\mathcal{O}$ and maintenances $\mathcal{R}$ that have not yet become available (see the definition of $\mathcal{A}_k$ below) for the JPSM-procedures at the current local time of their respective machines. The direct predecessors of o is the direct predecessor according to the job $j(o)$ and operations on the same machine for which a precedence relation is found during earlier passes of **UpdateHeads**. During DP these precedence relations are kept locally on the partial solutions and preserved during the expansions, see section 6.4.3.

For each machine m_k we define the following sets:

Algorithm 6.2 Update heads of normal operations

Input: An upper bound $\mathcal{UB}$
Output: A new lower bound $\mathcal{LB}$

UpdateHeads($\mathcal{UB}$)
  $\mathcal{LB} = 0$

  for all $k \in \mathcal{M}$ **do**
    InitData(k)
  for all $k \in \mathcal{M}$ **do**
    UpdateState(k,$\mathcal{UB}$)

  $\mathcal{M}^+ = \{1, \ldots, M\}$
  while $\mathcal{M}^+ \neq \emptyset$ **do**
    choose $k \in \mathcal{M}^+$ such that $t_k^{\text{req}} = \min_{k' \in \mathcal{M}^+} t_{k'}^{\text{req}}$
    if $t_k^{\text{req}} > \mathcal{UB}$ **then**
      $\mathcal{LB} = \mathcal{UB} + 1$
      break

    JPSMstep(k,$\mathcal{UB}$)
    if no $i \in \bar{I}_k$ with $p_k^+ > 0$ exists **then**
      $\mathcal{M}^+ = \mathcal{M}^+ - \{k\}$
      $\mathcal{LB} = \max \left\{\mathcal{LB}, \mathcal{LB}^k\right\}$
      if $\mathcal{LB} > \mathcal{UB}$ **then**
        $\mathcal{LB} = \mathcal{UB} + 1$
        break

  for all $k \in \mathcal{M}$ **do**
    SetHeads(k)

  return $\mathcal{LB}$

$\mathcal{A}_k = \left\{o \in \bar{I} \,\middle|\, \textsc{Indegree}_o = 0, \{w \in K_o^* \mid p_w^+ > 0\} = \emptyset, \tilde{r}_o < t_k, p_o^+ > 0\right\} \cup \{a_k\}$
  This is the set of operations of machine m_k which are available for processing at current time t_k.

$\mathcal{U}_k = \left\{o \in \bar{I} \,\middle|\, \textsc{Indegree}_o = 0, t_k < \tilde{r}_o\right\}$
  This the set of operations which are unavailable for processing on machine m_k at current time t_k.

$\mathcal{D}_k = \left\{o \in \bar{I} \,\middle|\, \textsc{Indegree}_o = 0, \{w \in K_o^* \mid p_w^+ > 0\} \neq \emptyset, t_k = \tilde{r}_o\right\} \cup$
$\left\{o \in \breve{I} \,\middle|\, \textsc{Indegree}_o = 0, t_k < h_k^{\max}, t_k = \tilde{r}_o\right\}$
  This is the set of operations o which are delayed until all operations in K_o^* are finished. As long as $o \in \mathcal{D}_k$, $\tilde{r}_o\tilde{r}_o$ is conceptually set to t_k, since this is certainly a valid head for operation o.

Algorithm 6.3 A single step in the JPSM procedure

Input: A machine index k
An upper bound $\mathcal{UB}$

JPSMstep(k,$\mathcal{UB}$)
$p^+_{o^*_k} = p^+_{o^*_k} - (t^{\text{req}}_k - t_k)$
$t_k = t^{\text{req}}_k$

if $p^+_{o^*_k} = 0$ **then**
$C_{o^*_k} = t_k$
$\mathcal{LB}^k = \max\left\{\mathcal{LB}^k, C_{o^*_k} + q_{o^*_k}\right\}$
$\mathcal{A}_k = \mathcal{A}_k - \{o^*_k\}$

if no $o \in \bar{I}_k$ with $p^+_o > 0$ exists **then**
return

for all $o \in \mathcal{D}_k$ with $\{o \in K^*_o \mid p^+_o > 0\} = \emptyset$ **do**
$\mathcal{D}_k = \mathcal{D}_k - \{o\}$
TryDelay(k, o,$\mathcal{UB}$)

UpdateState(k,$\mathcal{UB}$)

return

6

We define operation $o^*_k \in \mathcal{A}_k$ as the operation which is chosen to be processed at time t_k on machine m_k.

First **JPSMstep** processes operation o^*_k as long as possible, i.e., until it finishes or the next relevant time t^{req}_k is reached. When the operation o is finished it sets its finish time C_o, updates the lower bound, checks wether delayed operations have all their predecessors planned, and then calls the function **TryDelay** (algorithm 6.4) which tries to increase the head of an operation and propagates this head when it is definitely set and made available. Then **UpdateState** (algorithm 6.7) is called to check if further operations be made available and select a new operation o^*_k and update t^{req}_k.

The function **TryDelay** in algorithm 6.4 is applied to an operation o of which the head will be at least equal to the current time on the machine, i.e., $\tilde{r}_o \geq t_k$. If the head is equal to the current time it tries to increase the current head $\tilde{r}_o$ by first checking if the condition in inequality (6.3) applies. In this case o is added to $\mathcal{D}_k$. A maintenance is delayed when its currently determined earliest possible start time is larger than the current time $h^{\max}_k > t_k$. Otherwise the condition in inequality (6.2) is checked, and if it applies, then $\tilde{r}_o$ is updated and the operation o is added to $\mathcal{U}_k$. When none of the conditions are applicable o is added to $\mathcal{A}_k$ and $\tilde{r}_o$ is fixed and the heads of its successors are updated by

Algorithm 6.4 Try to delay an operation

Input: A machine index k
An operation o
An upper bound $\mathcal{UB}$

TryDelay$(k, o, \mathcal{UB})$
let $K_o^* \subset \bar{I}_k$ be the maximal set satisfying inequality (6.3)
if $K_o^* \neq \emptyset$ **then**
$\mathcal{D}_k = \mathcal{D}_k \cup \{o\}$
return

if $o \in \breve{I}_k$ **and** $h_k^{\max} > t_k$ **and** $h_k^{\min} < \infty$ **then**
$\mathcal{D}_k = \mathcal{D}_k \cup \{o\}$
return

let $\mathcal{Q}_k$ be a set of operations $w \in \mathcal{A}_k$ with $p_w^+ < p_w$ and $\tilde{r}_w + p_w > t_k$
if $\mathcal{Q}_k \neq \emptyset$ **then**
choose $w \in \mathcal{Q}_k$ such that $p_w + q_w = \max_{w' \in \mathcal{Q}_k} \{p_{w'} + q_{w'}\}$
if $t_k + p_o + p_w + q_w > \mathcal{UB}$ **then**
$\tilde{r}_o = \max\{\tilde{r}_o, \tilde{r}_w + p_w\}$
$\mathcal{U}_k = \mathcal{U}_k \cup \{o\}$
return

$\tilde{r}_o = t_k$
$\mathcal{A}_k = \mathcal{A}_k \cup \{o\}$
Propagate(k, o)

the function **Propagate**.

The **Propagate** function in algorithm 6.5 is called on an operation o when it is just made available in **TryDelay**. It propagates the head $\tilde{r}_o$ of operation o to its direct successors and if it is the last blocking predecessor it adds them to $\mathcal{U}$ of their respective machine and updates t^{req} of that machine. Recall that $m(o)$ gives the machine of operation o and we define $\textsc{Succ}_o$ and $\textsc{Pred}_o$ as the set of all predecessors and successors of o, respectively. If $o \in \breve{I}_k$ the values of $h_k^{\min}$ and $h_k^{\max}$ are updated. When there are no maintenances left they are set to ∞ otherwise all available and fully scheduled normal operations are used to determine $h_k^{\max}$. If $o \in I_k$ the value of $h_k^{\max}$ is updated when o can finish before $h_k^{\max}$.

Now we look at the initializing functions **InitData** in algorithm 6.6 and **UpdateState** in algorithm 6.7.

InitData initializes all basic data for machine k and sets the current time to 0. **UpdateState** considers all operations of machine k without a direct predecessor and their head equal to the current time and tries to increase its head using **TryDelay**. The same holds for a delayed maintenance when $t_k = h_k^{\max}$.

Algorithm 6.5 Propagate an finalized head

Input: A machine index k
An operation o

Propagate(k, o)
  for all $w \in \textsc{Succ}_o$ **do**
    $\tilde{r}_w = \max\{\tilde{r}_w, t_k + p_o\}$
    $\textsc{Indegree}_o = \textsc{Indegree}_o - 1$
    if $\textsc{Indegree}_o = 0$ **then**
      $\mathcal{U}_{m(w)} = \mathcal{U}_{m(w)} \cup \{w\}$
      if $m(w) \neq k$ **and** $\tilde{r}_w < t^{\text{req}}_{m(w)}$ **then**
        $t^{\text{req}}_{m(w)} = \tilde{r}_w$

  if $o \in \breve{I}_k$ **then**
    $h^{\min}_k = h^{\max}_k = \infty$
    if $\textsc{Succ}_o \cap \breve{I}_k \neq \emptyset$ **then**
      $h^{\min}_k = t_k + p_o$
      let $\mathcal{R}_k$ be the set $(\mathcal{A}_k \cap I_k) \cup \{v \in I_k \mid p^+_v = 0\}$
      let $\mathcal{F}_k$ be the set $\left\{w \in \mathcal{R}_k \,\middle|\, h^{\min}_k + p_w + q_w \leq \mathcal{UB}\right\}$
      if $\mathcal{F}_k \neq \emptyset$ **then**
        $h^{\max}_k = h^{\min}_k + \min_{w \in \mathcal{F}_k} p_w$

  if $o \in I_k$ **then**
    $h^{\max}_k = \min\left\{h^{\max}_k, \max\left\{h^{\min}_k, t_k\right\} + q_o\right\}$

  return

Algorithm 6.6 Initializes data on a machine

Input: A machine index k

InitData(k)
  for all $o \in \bar{I}$ **do**
    $p^+_o = p_o$
    $\textsc{Indegree}_o = |\textsc{Pred}_o|$
    $K^*_o = \emptyset$
    $\tilde{r}_o = r_o$

    $\mathcal{U}_k = \left\{o \in \bar{I} \,\middle|\, \textsc{Indegree}_o = 0\right\}$
    $\mathcal{A}_k = \{a_k\}$
    $\mathcal{D}_k = \emptyset$
    $t_k = t^{\text{req}}_k = \mathcal{LB}^k = 0$
    $h^{\min}_k = h^{\max}_k = \infty$
  return

Algorithm 6.7 Updates the state on a machine

Input: A machine index k
An upper bound $\mathcal{UB}$

UpdateState(k)
  for all $o \in \mathcal{U}_k$ with $\tilde{r}_o = t_k$ **do**
    $\mathcal{U}_k = \mathcal{U}_k - \{o\}$
    TryDelay($k, o, \mathcal{UB}$)

  if $h_k^{\max} = t_k$ **then**
    $h_k^{\min} = h_k^{\max} = \infty$
    for $o \in \mathcal{D}_k \cap \breve{I}$ **do**
      $\mathcal{D}_k = \mathcal{D}_k - \{o\}$
      TryDelay($k, o, \mathcal{UB}$)

  choose $o_k^* \in \mathcal{A}_k$ such that $q_{o_k^*} = \max_{o \in \mathcal{A}_k} q_o$
  $t_k^{\text{req}} = \min \left\{ t_k + p_{o_k^*}^+,\ \min_{o \in \mathcal{U}_k} \tilde{r}_o,\ h_k^{\max} \right\}$

Algorithm 6.8 Sets the heads of the operations to the new obtained heads

Input: A machine index k

SetHeads(k)
  for all $o \in I$ **do**
    $r_o = \tilde{r}_o$
  return

6

If no improvement is possible the operation is made available in **TryDelay**. Finally the operation to be processed (o_k^*) is selected and t_k^{req} is set.

Note that when no operation is available dummy operation a_k is selected for processing. If it is also the case that $\mathcal{U}_k = \emptyset$, t_k^{req} is set to ∞, if no inconsistency is detected it will be shortened when a job is added to $\mathcal{U}_k$ due to the fact that its final predecessor finishes.

Finally the heads are updated by **SetHeads** function in algorithm 6.8.

6.4.3 Dynamic bounding for the JSSPM

To incorporate this bounding into DP we update the heads and tails for each partial solution. Conceptually we can ignore all scheduled operations and maintenances and can use the left uptime u and the remaining operations for each machine to determine a possible better lower bound $\mathcal{N}_k$ for the remaining maintenances. The remaining operations can be derived from S and the values of $\vec{u}$ can be employed to calculate the used uptime $U_k - u$, which can be seen as an extra operation that will be included to determine $\mathcal{N}_k$. As with the bound for the

JSSP we can initialize the heads of the remaining operations to $\alpha(\varsigma_S, o) - p(o)$. Finally, the values of u are also used to initialize $h_k^{\min}$ and $h_k^{\max}$ for the first iteration of the forward JPSM. If $u = U_k$ for a machine k, $h_k^{\min}$ is initialized as normal, the processing conditions of the machine are fully restored, as in the beginning of the planning period. However, when a machine is not fresh, $u < U_k$ for that machine, $h_k^{\min}$ and $h_k^{\max}$ are set to 0 as the maintenance can be planned immediately.

During the bounding procedure precedence relations, specific to a partial solution, are generated. However, precedence relations regarding maintenances cannot be used during DP as the number of maintenances used by the bounding is an estimate and the maintenances used during bounding cannot be mapped to the maintenances scheduled by DP. Although the heads and tails can be calculated based on a state, in practice it is more convenient to save the calculated heads and tails with the partial solutions, they can be seen as bookkeeping variables β. As with the precedence relations the heads and tails cannot be saved for maintenances for the same reasoning.

6.5 JSSPM instances

For the Job Shop Scheduling Problem with scheduled Maintenances there are no instances known in the literature. To obtain multiple problem instances for the JSSPM we describe here a set of possible conversions of a regular instance for the JSSP to an instance of the JSSPM.

For the JSSPM we have to come up with two numbers per machine i, a maximum uptime without maintenance U_i and the duration of the maintenance downtime D_i. For an instance to have a feasible solution the maximum uptime for a machine should be at least as long as the longest operation on that machine. For both U_i and D_i we base its value on one of two properties of the original JSSP instance. The maximum duration of an operation or the sum of the duration of the operations on a machine.

Let us define these values first. Let S_i be the sum of the duration all operations on machine i, and let S be the maximum of S_i over all machines. Let M_i be the maximum of the durations of all operations on machine i, and let M be the maximum of M_i over all machines, that is the maximum duration of any operation. Thus

$$\begin{aligned} S_i &= \sum_{o \in \mathcal{O} | m(o) = i} p(o) \\ S &= \max_{i \in \mathcal{M}} S_i \\ M_i &= \max_{o \in \mathcal{O} | m(o) = i} p(o) \\ M &= \max_{i \in \mathcal{M}} M_i = \max_{o \in \mathcal{O}} p(o) \end{aligned}$$

Now we create the instances as follows. For both uptime U and downtime D we determine a factor f_U and f_D and we multiply by this factor for each machine with S_i, S, M_i or M, we call these four possibilities:

nh sum	Non-homogeneous sum (S_i)
h sum	Homogeneous sum (S)
nh max	Non-homogeneous max (M_i)
h max	Homogeneous max (M)

We take the ceiling of the resulting value. Furthermore, the minimum value for D_i is set to 1 and the minimum value for U_i is set to M_i for the *non-homogeneous* variant and to M for the *homogeneous* variant independent of the value of f_U and f_D. Naturally, $f_U < 1$ would result in $f_U = 1$ when using one of the *max* variants for U and $f_U \geq 1$ would result in the original JSSP instance, since all operations can be done without maintenance.

This leads to the following formulas:

$$U_i = \begin{cases} \max\{M_i, \lceil f_U S_i \rceil\} & \text{nh sum} \\ \max\{M, \lceil f_U S \rceil\} & \text{n sum} \\ \max\{M_i, \lceil f_U M_i \rceil\} & \text{nh max} \\ \max\{M, \lceil f_U M \rceil\} & \text{n max} \end{cases}$$

$$D_i = \begin{cases} \max\{1, \lceil f_D S_i \rceil\} & \text{nh sum} \\ \max\{1, \lceil f_D S \rceil\} & \text{n sum} \\ \max\{1, \lceil f_D M_i \rceil\} & \text{nh max} \\ \max\{1, \lceil f_D M \rceil\} & \text{n max} \end{cases}$$

We describe an instance of the JSSPM as follows $\mathbf{ft06|nh|sum|\frac{1}{3}|h|max|\frac{3}{4}}$, where the first element is the original JSSP instance, parts 2–4 and 5–7 describe the *(non–)homogeneity*, *sum* or *max* and the factor for uptime U and downtime D, respectively.

We generated 288 instances using all possible combinations given in table 6.2.

Original Instance		Uptime U	Downtime D
• ft06	• la03	• h/nh|max|1	• h/nh|max|1
• la01	• la04	• h/nh|max|$\frac{3}{2}$	• h/nh|max|$\frac{1}{3}$
• la02	• la05	• h/nh|sum|$\frac{1}{3}$	• h/nh|sum|$\frac{1}{5}$
		• h/nh|sum|$\frac{2}{3}$	

Table 6.2: *Combinations to generate* 288 *JSSPM instances*

6.6 Comparing DP to MIP

We used the DP algorithm described in section 6.3 to find solutions for these instances. We started without setting any bound and expanding only 1000 partial solutions in each stage by setting $H = 1000$. To select the most promising solutions to expand the lower bound on the completion cost, as described in section 6.4, is used, similar to the selection done for the JSSP. When the width H did not limit the state space we found the optimal solution, or when no solution is found the previous found solution is proven to be optimal. If a solution was found but width H was limitative we repeated the algorithm, setting a new upper bound equal to the value of the found solution minus one. When no solution is found, width H is increased by a factor 10 until at most $H = 10^6$. However, when no solution was found in the first run with no bound a fictive bound of 3000 was used in the next run. The total results are given in table A.5 (pages 142–151).

For the 46 instances where the DP algorithm could not prove optimality we used the procedure described in section 5.2.4 to find a lower bound. As start value we used the best known value from the first procedure minus 400. The state space is bounded with width $H = 10^5$ for these runs. Results of this procedure can be found in table A.6 (page 152).

To compare the performance of our DP algorithm we solved the MIP model described section 6.2 in using *Gurobi 5.6.3*. The following parameters where given to the solver to solve the MIP model for all instances, using a time limit of 2 hours:

TimeLimit	7200
Method	2
Presolve	2
GomoryPasses	1

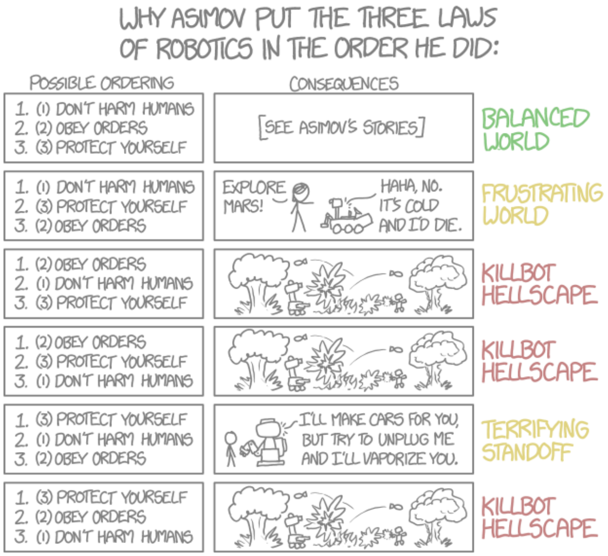
6

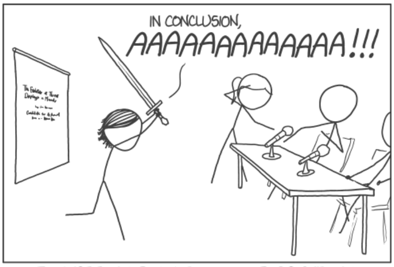
These parameters are found by running the automatic configuration tool of Gurobi on the *ft06* instances for 2 hours. We combined the parameters of the best parameter sets returned by the tuning tool which proved to be even better according to the tuning tool. In table A.7 these results are combined with the summary of the results from the DP algorithm given in tables A.5 and A.6. Note that Gurobi was able to use both CPU cores while DP only used a single core.

We can see that the small *ft06* instances are all solved by MIP and DP, although DP outperforms MIP on all but 4 of the 48 instances, even when we look at the first time MIP found the optimal solution. For the *la* instances we see that DP finds the optimal solution for 194 of the 240 instances while using the MIP model only the optimal value for 54 of these 194 instances is found and for only 3 the optimality was proven. We should note that for 12 of these instances DP used more than 7200 seconds but never more than 10000.

For most of the remaining 46 instances, DP algorithm spent more than 7200 or even 14400 seconds. For one of these instances optimality was proven using the MIP, for 12 other instances a better solution was found by solving the MIP than by applying the DP algorithm. For two of these 12 instances also a better lower bound was found using the MIP.

We clearly see that the instances where the uptime is defined by h/nh|sum|$\frac{1}{3}$ (see table 6.2) seem to be harder to solve than the instances which are defined by h/nh|max|1, h/nh|max|$\frac{3}{2}$ or h/nh|sum|$\frac{2}{3}$. We suspect that the instances where the uptime is defined by h/nh|max|1, h/nh|max|$\frac{3}{2}$ are easier because between almost all operations a maintenance seems to be needed and for h/nh|sum|$\frac{2}{3}$ only a single maintenance is needed.


WHY ASIMOV PUT THE THREE LAWS OF ROBOTICS IN THE ORDER HE DID:
POSSIBLE ORDERING
CONSEQUENCES
1. (1) DON'T HARM HUMANS
2. (2) OBEY ORDERS
3. (3) PROTECT YOURSELF
[SEE ASIMOV'S STORIES]
BALANCED WORLD
1. (1) DON'T HARM HUMANS
2. (3) PROTECT YOURSELF
3. (2) OBEY ORDERS
EXPLORE MARS!
HAHA, NO. IT'S COLD AND I'D DIE.
FRUSTRATING WORLD
1. (2) OBEY ORDERS
2. (1) DON'T HARM HUMANS
3. (3) PROTECT YOURSELF
KILLBOT HELLSCAPE
1. (2) OBEY ORDERS
2. (3) PROTECT YOURSELF
3. (1) DON'T HARM HUMANS
KILLBOT HELLSCAPE
1. (3) PROTECT YOURSELF
2. (1) DON'T HARM HUMANS
3. (2) OBEY ORDERS
I'LL MAKE CARS FOR YOU, BUT TRY TO UNPLUG ME AND I'LL VAPORIZE YOU.
TERRIFYING STANDOFF
1. (3) PROTECT YOURSELF
2. (2) OBEY ORDERS
3. (1) DON'T HARM HUMANS
KILLBOT HELLSCAPE



THE BEST THESIS DEFENSE IS A GOOD THESIS OFFENSE.

Concluding Remarks

Dynamic Programming over sets exists for over half a century, with the Dynamic Programming algorithm for the Traveling Salesman Problem as its prime example. Although Dynamic Programming is often viewed as impractical to solve NP-hard problems, it still provides the algorithm with the lowest time complexity to solve the Traveling Salesman Problem.

In this dissertation we extend this algorithm to the Vehicle Routing Problem and to the Job Shop Scheduling Problem. The extension of the Dynamic Programming algorithm for the Traveling Salesman Problem to the Vehicle Routing Problem is fairly straightforward, since the Vehicle Routing Problem is very similar to the Traveling Salesman Problem. The extension to the Job Shop Scheduling Problem is more complicated, since the Job Shop Scheduling Problem is a *min–max* problem, in contrast to the *min–sum* objective of the Vehicle Routing Problem and Traveling Salesman Problem. We show how to use the principle of Dynamic Programming over sets to solve such a *min–max* objective. Dynamic Programming for the Job Shop Scheduling Problem offers currently, up to our knowledge, the algorithm with the lowest time complexity to solve the Job Shop Scheduling Problem.

For these problems we show how the Dynamic Programming algorithm may be altered to solve extensions to the problem, while keeping the guarantee to find the optimal solution. For the Vehicle Routing Problem we illustrate this for a large range of known and lesser known extensions. For the Job Shop Scheduling Problem we show in detail how to add maintenance creating a new problem, the Job Shop Scheduling Problem with scheduled Maintenances.

For this new problem, the Job Shop Scheduling Problem with scheduled Maintenances, we created a new Mixed-Integer Programming formulation to be able to evaluate our Dynamic Programming algorithm. As this is a new problem, not yet found in the literature, we created a general way to use existing Job Shop Scheduling Problem benchmark instances to create new instances of the Job Shop Scheduling Problem with scheduled Maintenances with different characteristics. This procedure is limited, in the sense that the same proportional relation between the operations on a machine and the maintenance plan of the machine exists. However, it can be used to create a diverse set of benchmark instances of the Job Shop Scheduling Problem with scheduled Maintenances based on existing benchmark instances of the Job Shop Scheduling Problem.

Our computational results show that the Dynamic Programming algorithm is very competitive compared to a state of the art Mixed-Integer Programming solver applied to our Mixed-Integer Programming formulation.

Although Dynamic Programming provides the algorithm with the best runtime complexity known to solve these problems to optimality, from a practical point of view it is hard to use as the time complexity, as well as the memory complexity, are exponential. To increase the practicality of the Dynamic Programming algorithms, we show how to, on one hand add bounding to these Dynamic Programming algorithms, preserving the optimality, and on the other hand how Dynamic Programming algorithms can be used as a heuristic. Although this works quite well, it is largely dependent on a good lower bound on, or an estimation of, the cost of all possible completions of the current partial solution, respectively. With a good estimate on the completion for each partial solution, the Dynamic Programming algorithm is able to provide good solutions, while using only a very narrow state space. Although, similar in concept to *beam search* applied to the exploration of the solution space of a problem, our ideas apply to the state space of a Dynamic Programming algorithm.

We use a Dynamic Programming algorithm as basis to create an algorithm which finds all optimal solutions of a given problem. As far as we know, finding all optimal solutions to the optimization problems described in this dissertation, has not yet been considered. We notice that the number of optimal solutions for small Job Shop Scheduling Problem benchmark instances can greatly differ. We could not find a relation between the number of optimal solutions and the observed effort to find a single optimal solution.

Although we achieved nice results with our Dynamic Programming algorithms for the Vehicle Routing Problem and the Job Shop Scheduling Problem there are still plenty of areas where it may be improved. For example, the Dynamic Programming algorithms can be used inside a larger algorithm framework, other bounding algorithms can be used or developed, and it may be possible to change the Dynamic Programming algorithm itself to achieve a better practical performance.

The fact that a lot of different extensions can be incorporated into the Dynamic Programming algorithm for the Vehicle Routing Problem makes it a general framework to solve rich Vehicle Routing Problems. We think that a good, and fast, lower bound on the cost can greatly help to improve the quality of the solutions found by heuristic versions of the Dynamic Programming algorithm. Also, using Dynamic Programming as pricing instrument in a column generation framework can provide a flexible way to solve rich Vehicle Routing Problems. The state space of the Dynamic Programming algorithm is limited by the fact that Dynamic Programming is used only to create a single route in this case. We can limit the Dynamic Programming state space further by bounding or using a heuristic. Than, given a good lower bound or estimator, this may prove to be a powerful technique to solve rich Vehicle Routing Problems.

For the Job Shop Scheduling Problem we created a Dynamic Programming algorithm for a *min–max* optimization problem. It may be possible to use the basis of this algorithm to create Dynamic Programming algorithms for other *min–max* optimization problems.

Finally, in this dissertation the Dynamic Programming algorithms are executed stage by stage. It may be worth to try to expand the partial solutions using a less predefined pattern. For example, expand the partial solution with the lowest lower bound on cost of any completion, disregarding the stage of the partial solution. This is similar to *best–first–search* in branch and bound. This may result in more effort in the beginning of the state space, compared to limiting the number of expanded solutions, but preserves the guarantee of optimality. The same state space would be evaluated as running the Dynamic Programming algorithm with the optimal value as upper-bound.

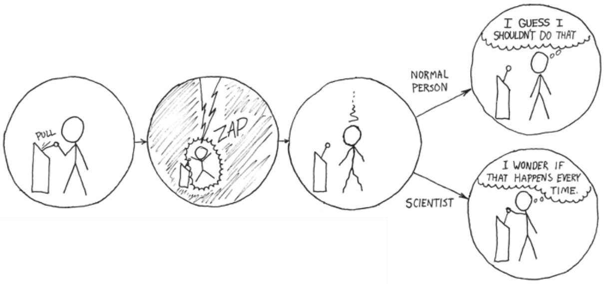
Furthermore, when a partial solution has a good lower bound, it may be the case that it can be expanded keeping the same lower bound for one of the expansions. This would create a kind of *depth–first–search* within the Dynamic Programming state space. The risk is that partial solutions get expanded which would otherwise be dominated. Therefore, a tradeoff may have to be found in this case.

It may be that it is possible to prevent the expansion of partial solutions that would otherwise be dominated. For example, the bound used in this dissertation for the Job Shop Scheduling Problem, which depends on the state variables used to compare for domination, may prevent the extension of partial solutions that would be dominated. This is a result of the fact that the lower bound of a dominated solution cannot be lower than the lower bound of its dominating solution. A downside for the Job Shop Scheduling Problem is that the quality of the bound used in this dissertation depends on the given upper bound. Also, when a solution is found and hence a new upper bound becomes available then the lower bounds of all not yet expanded solutions should be recalculated.

Another idea is to start Dynamic Programming using a lower bound of the whole problem as an upper bound and expand all partial solutions until all of them would be bounded, i. e., the lower bound on each partial solution is higher than the used upper bound. After that, the upper bound should be increased by one and the current lower bounds of all non-expanded partial solutions should be updated. This process can be repeated until the optimal solution is found.

Concluding, there are still numerous interesting research directions, which may lead to further improvements of the algorithms described in this dissertation.


WHY ARE PEOPLE SO COMFORTABLE HANDING GOOGLE AND FACEBOOK ALL THIS CONTROL OVER OUR LIVES?
I DUNNO.
OUR SPECIES BUILT THOUSANDS OF NUCLEAR WEAPONS, SCATTERED THEM AROUND THE PLANET, AND THEN MOVED ON TO OTHER THINGS.
MAYBE IT'S BEST TO ACCEPT THAT SOME OF THIS BIG-PICTURE PLANNING IS JUST HAPPENING ON AUTOPILOT.



Appendix A

Computational Results

All results are generated on a machine with a dual core 3.00 Ghz CPU, with 16GB of memory. It has a 64-bit version of *windows 7* installed. All of our own implementations are written in C++ and all MIP/LP calculations are done with *Gurobi 5.6.3*.

Overview of tables

Instance	UB	{H = 10, E = 10}			{H = 25, E = 25}			{H = 50, E = 25}			{H = 75, E = 25}		
		cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)
A-n32-k5	**784**	815	4.0	1	-	-	3	**784**	0	5			
A-n33-k5	**661**	**661**	0	1									
A-n33-k6	**742**	750	1.1	1	**742**	0	3						
A-n34-k5	**778**	792	1.8	1	-	-	3	-	-	6	791	1.7	9
A-n36-k5	**799**	840	5.1	1	807	1.0	4	-	-	8	-	-	12
A-n37-k5	**669**	670	0.1	1	-	-	5	-	-	11	-	-	14
A-n37-k6	**949**	1019	7.4	1	970	2.2	5	966	1.8	9	-	-	13
A-n38-k5	**730**	787	7.8	1	762	4.4	5	-	-	10	759	4.0	13
A-n39-k5	**822**	861	4.7	1	851	3.5	7	-	-	13	-	-	20
A-n39-k6	**831**	846	1.8	1	-	-	6	-	-	12	842	1.3	18
A-n44-k6	**937**	947	1.1	2	946	1.0	9	944	0.7	17	-	-	26
A-n45-k6	**944**	-	-	2	1062	12.5	9	975	3.3	17	-	-	25
A-n45-k7	**1146**	1174	2.4	2	-	-	10	1161	1.3	20	-	-	26
A-n46-k7	**914**	922	0.9	2	921	0.8	11	920	0.7	19	-	-	29
A-n48-k7	**1073**	1133	5.6	3	1075	0.2	12	**1073**	0	23			
A-n53-k7	**1010**	1046	3.6	4	1028	1.8	19	-	-	37	-	-	55
A-n54-k7	**1167**	1252	7.3	4	1190	2.0	18	1189	1.9	37	1183	1.4	58
A-n55-k9	**1073**	1109	3.4	4	-	-	20	1108	3.3	41	1107	3.2	62
A-n60-k9	**1354**	1463	8.1	5	1446	6.8	29	1373	1.4	53	-	-	79
A-n61-k9	**1034**	1075	4.0	5	-	-	28	1066	3.1	52	-	-	78
A-n62-k8	**1288**	1391	8.0	6	1354	5.1	33	1340	4.0	64	1321	2.6	98
A-n63-k9	**1616**	1687	4.4	7	1637	1.3	35	-	-	70	-	-	102
A-n63-k10	**1314**	1396	6.2	7	1345	2.4	35	-	-	68	1326	0.9	102
A-n64-k9	**1401**	1519	8.4	7	1466	4.6	35	1457	4.0	70	-	-	111
A-n65-k9	**1174**	1206	2.7	7	-	-	36	1178	0.3	68	-	-	106
A-n69-k9	**1159**	1189	2.6	9	1174	1.3	48	-	-	95	-	-	140
A-n80-k10	**1763**	1850	4.9	14	1832	3.9	78	1823	3.4	152	1813	2.8	228

Table A.1: *Results on CVRP instances*

Instance	UB	{H = 10, E = 10}			{H = 25, E = 25}			{H = 50, E = 25}			{H = 75, E = 25}		
		cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)
B-n31-k5	**672**	681	1.3	0	**672**	0	2						
B-n34-k5	**788**	825	4.7	1	**788**	0	3						
B-n35-k5	**955**	971	1.7	1	-	-	3	-	-	6	-	-	9
B-n38-k6	**805**	835	3.7	1	831	3.2	5	817	1.5	9	-	-	13
B-n39-k5	**549**	633	15.3	1	565	2.9	5	-	-	10	-	-	16
B-n41-k6	**829**	953	15.0	1	923	11.3	7	869	4.8	12	-	-	17
B-n43-k6	**742**	792	6.7	2	769	3.6	8	-	-	15	-	-	22
B-n44-k7	**909**	934	2.8	2	-	-	8	933	2.6	16	-	-	24
B-n45-k5	**751**	844	12.4	2	826	10.0	9	-	-	15	-	-	22
B-n45-k6	**678**	720	6.2	2	716	5.6	8	687	1.3	15	-	-	21
B-n50-k7	**741**	841	13.5	3	787	6.2	12	-	-	24	781	5.4	36
B-n50-k8	**1312**	1376	4.9	3	1349	2.8	13	1346	2.6	26	-	-	39
B-n51-k7	**1032**	1264	22.5	3	1129	9.4	12	1123	8.8	23	1094	6.0	32
B-n52-k7	**747**	789	5.6	3	763	2.1	16	755	1.1	31	-	-	44
B-n56-k7	**707**	741	4.8	4	-	-	20	-	-	39	722	2.1	57
B-n57-k7	**1153**	1408	22.1	4	1337	16.0	20	-	-	37	1227	6.4	52
B-n57-k9	**1598**	1718	7.5	5	1687	5.6	23	1635	2.3	41	1619	1.3	62
B-n63-k10	**1496**	1560	4.3	6	1551	3.7	30	1548	3.5	59	1542	3.1	86
B-n64-k9	**861**	944	9.6	6	865	0.5	32	-	-	66	-	-	91
B-n66-k9	**1316**	1386	5.3	6	1358	3.2	36	-	-	69	-	-	104
B-n67-k10	**1032**	1143	10.8	7	1106	7.2	39	1103	6.9	76	1099	6.5	110
B-n68-k9	**1272**	1354	6.4	8	1327	4.3	42	1314	3.3	82	-	-	119
B-n78-k10	**1221**	1304	6.8	13	1296	6.1	65	1260	3.2	129	1255	2.8	187

Table A.1: *Results on CVRP instances (continued)*

Instance	UB	{H = 10, E = 10}			{H = 25, E = 25}			{H = 50, E = 25}			{H = 75, E = 25}		
		cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)
P-n16-k8	**450**	**450**	0	0									
P-n19-k2	**212**	**212**	0	0									
P-n20-k2	**216**	**216**	0	0									
P-n21-k2	**211**	**211**	0	0									
P-n22-k2	**216**	**216**	0	0									
P-n22-k8	**603**	631	4.6	0	604	0.2	1	-	-	1	-	-	1
P-n23-k8	**529**	**529**	0	0									
P-n40-k5	**458**	467	2.0	2	463	1.1	7	-	-	14	-	-	21
P-n45-k5	**510**	516	1.2	2	**510**	0	11						
P-n50-k7	**554**	585	5.6	3	565	2.0	13	563	1.6	28	-	-	42
P-n50-k8	**631**	670	6.2	3	662	4.9	14	657	4.1	23	-	-	36
P-n50-k10	**696**	758	8.9	3	722	3.7	14	-	-	27	703	1.0	43
P-n51-k10	**741**	904	22.0	3	824	11.2	13	755	1.9	27	-	-	43
P-n55-k7	**568**	603	6.2	4	577	1.6	20	-	-	41	574	1.1	62
P-n55-k8	576	595	3.3	4	590	2.4	22	589	2.3	43	586	1.7	66
P-n55-k10	**694**	718	3.5	5	-	-	21	713	2.7	43	698	0.6	68
P-n55-k15	**989**	-	-	2	-	-	14	-	-	26	1034	4.6	39
P-n60-k10	**744**	774	4.0	5	759	2.0	27	751	0.9	58	746	0.3	86
P-n60-k15	**968**	1002	3.5	5	988	2.1	27	-	-	57	-	-	82
P-n65-k10	**792**	862	8.8	7	807	1.9	39	-	-	81	-	-	121
P-n70-k10	**827**	905	9.4	9	892	7.9	48	858	3.7	91	849	2.7	145
P-n76-k4	**593**	611	3.0	11	598	0.8	81	595	0.3	199	**593**	0	305
P-n76-k5	**627**	640	2.1	11	-	-	72	-	-	148	638	1.8	225
P-n101-k4	**681**	691	1.5	27	-	-	219	-	-	455	-	-	678
F-n45-k4	**724**	840	16.0	2	733	1.2	8	-	-	16	-	-	25
F-n72-k4	**237**	261	10.1	8	257	8.4	43	-	-	86	-	-	127
F-n135-k7	**1162**	1334	14.8	58	1260	8.4	326	-	-	650	1244	7.1	957

Table A.1: *Results on CVRP instances (continued)*

Instance	UB	{H = 10, E = 10}			{H = 25, E = 25}			{H = 50, E = 25}			{H = 75, E = 25}		
		cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)	cost	gap (%)	CPU (s)
E-n13-k4	**247**	**247**	0	0									
E-n22-k4	**375**	**375**	0	0									
E-n23-k3	**569**	577	1.4	0	-	-	1	**569**	0	1			
E-n30-k3	**534**	575	7.7	0	-	-	2	553	3.6	3	540	1.1	4
E-n31-k7	**379**	**379**	0	1									
E-n33-k4	**835**	837	0.2	1	**835**	0	3						
E-n51-k5	**521**	**521**	0	3									
E-n76-k7	**682**	708	3.8	11	696	2.1	62	688	0.9	135	-	-	214
E-n76-k8	**735**	781	6.3	12	736	0.1	65	-	-	143	-	-	212
E-n76-k10	**830**	859	3.5	12	835	0.6	63	-	-	129	-	-	208
E-n76-k14	**1021**	1167	14.3	11	1116	9.3	61	1083	6.1	123	1075	5.3	184
E-n101-k8	**815**	859	5.4	26	847	3.9	158	846	3.8	314	845	3.7	467
E-n101-k14	**1067**	1153	8.1	29	1122	5.2	168	1101	3.2	343	-	-	514
G-n262-k25	5685	5941	4.5	500	-	-	2907	5785	1.8	5891	-	-	8748
M-n101-k10	**820**	874	6.6	25	847	3.3	145	837	2.1	276	833	1.6	427
M-n121-k7	**1034**	1072	3.7	35	-	-	228	-	-	506	1055	2.0	767
M-n151-k12	**1015**	1059	4.3	95	1056	4.0	552	1046	3.1	1109	-	-	1632
M-n200-k16	**1274**	1521	19.4	205	1400	9.9	1215	1370	7.5	2435	1333	4.6	3695
M-n200-k17	**1275**	1399	9.7	226	-	-	1311	1326	4.0	2680	1321	3.6	3987
att-n48-k4	**40002**	40625	1.6	2	40101	0.2	11	-	-	23	-	-	35
bayg-n29-k4	**2050**	2055	0.2	1	**2050**	0	2						
bays-n29-k5	**2963**	3236	9.2	0	2978	0.5	2	-	-	3	-	-	4
dantzig-n42-k4	**1142**	1212	6.1	2	1211	6.0	8	-	-	14	-	-	21
fri-n26-k3	**1353**	1358	0.4	0	-	-	1	-	-	2	-	-	3
gr-n17-k3	**2685**	2838	5.7	0	2769	3.1	0	**2685**	0	0			
gr-n21-k3	**3704**	3880	4.8	0	3755	1.4	0	**3704**	0	1			
gr-n24-k4	**2053**	2193	6.8	0	2154	4.9	1	-	-	2	2080	1.3	2
gr-n48-k3	**5985**	6429	7.4	3	6401	7.0	12	-	-	22	-	-	35
hk-n48-k4	**14749**	15208	3.1	3	14844	0.6	12	-	-	23	-	-	35
swiss-n42-k5	**1668**	1753	5.1	1	1737	4.1	8	1717	2.9	14	-	-	21
ulysses-n16-k3	**7965**	8222	3.2	0	-	-	0	-	-	0	-	-	0
ulysses-n22-k4	**9179**	9748	6.2	0	9344	1.8	0	-	-	1	9301	1.3	2

Table A.1: *Results on CVRP instances (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
la01	10	5		*666*	667	0.2	5	1	1	666	0	2	1	1
la02	10	5		*655*	659	0.6	3	0	1	655	0	2	0	1
la03	10	5		*597*	614	2.8	4	0	1	597	0	2	1	1
la04	10	5		*590*	590	0	5	0	1	-	-	-	-	-
la05	10	5		*593*	593	0	2	0	1	-	-	-	-	-
la06	15	5		*926*	926	0	4	0	1	-	-	-	-	-
la07	15	5		*890*	890	0	3	0	1	-	-	-	-	-
la08	15	5		*863*	863	0	4	0	1	-	-	-	-	-
la09	15	5		*951*	951	0	3	1	1	-	-	-	-	-
la10	15	5		*958*	958	0	2	0	1	-	-	-	-	-
la11	20	5		*1222*	1222	0	3	1	1	-	-	-	-	-
la12	20	5		*1039*	1039	0	3	1	1	-	-	-	-	-
la13	20	5		*1150*	1150	0	3	1	1	-	-	-	-	-
la14	20	5		*1292*	1292	0	3	0	1	-	-	-	-	-
la15	20	5		*1207*	1207	0	8	4	1	-	-	-	-	-
la16	10	10		*945*	988	4.6	5	1	1	964	2.0	2	2	1
la17	10	10		*784*	793	1.1	4	0	1	784	0	3	2	1
la18	10	10		*848*	880	3.8	5	0	1	849	0.1	2	2	1
la19	10	10		*842*	863	2.5	3	0	1	848	0.7	2	2	1
la20	10	10		*902*	949	5.2	4	1	1	902	0	3	1	1
la21	15	10		*1046*	1198	14.5	5	2	1	1107	5.8	7	32	2
la22	15	10		*927*	1041	12.3	6	2	1	977	5.4	4	17	2
la23	15	10		*1032*	1067	3.4	6	3	1	1051	1.8	2	8	2
la24	15	10		*935*	973	4.1	9	2	1	953	1.9	3	11	2
la25	15	10		*977*	1070	9.5	3	2	1	1024	4.8	4	17	2

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
la26	20	10		*1218*	1272	4.4	3	2	1	-	-	1	9	2
la27	20	10		*1235*	1363	10.4	4	4	1	1313	6.3	3	31	2
la28	20	10		*1216*	1313	8.0	6	5	1	1234	1.5	6	72	2
la29	20	10		*1152*	1315	14.1	5	4	1	1257	9.1	3	29	2
la30	20	10		*1355*	1439	6.2	4	3	1	1366	0.8	4	36	2
la31	30	10		*1784*	1787	0.2	5	12	1	1784	0	2	61	4
la32	30	10		*1850*	1883	1.8	7	22	2	1850	0	4	128	4
la33	30	10		*1719*	1721	0.1	8	24	2	1719	0	2	59	4
la34	30	10		*1721*	1734	0.8	5	11	1	1721	0	3	92	4
la35	30	10		*1888*	1891	0.2	4	10	1	1888	0	2	59	3
la36	15	15		*1268*	1363	7.5	4	3	1	1337	5.4	2	12	2
la37	15	15		*1397*	1481	6.0	4	2	1	1461	4.6	4	28	2
la38	15	15		*1196*	1360	13.7	6	4	1	1250	4.5	5	35	2
la39	15	15		*1233*	1367	10.9	5	4	1	1328	7.7	3	20	2
la40	15	15		*1222*	1307	7.0	7	6	1	1297	6.1	2	13	2
orb01	10	10		*1059*	1107	4.5	3	0	1	1060	0.1	2	2	1
orb02	10	10		*888*	944	6.3	4	0	1	908	2.3	2	3	1
orb03	10	10		*1005*	1083	7.8	3	0	1	1036	3.1	2	3	1
orb04	10	10		*1005*	1044	3.9	6	2	1	1022	1.7	2	2	1
orb05	10	10		*887*	937	5.6	4	0	1	898	1.2	4	5	1
orb06	10	10		*1010*	1093	8.2	3	0	1	1033	2.3	3	3	1
orb07	10	10		*397*	506	27.5	2	0	1	405	2.0	2	2	1
orb08	10	10		*899*	947	5.3	4	0	1	939	4.4	2	2	1
orb09	10	10		*934*	954	2.1	4	0	1	942	0.9	2	2	1
orb10	10	10		*944*	1012	7.2	4	2	1	984	4.2	2	3	1

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	*H* = 10					*H* = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
ft06	6	6		*55*	55	0	2	0	1	-	-	-	-	-
ft10	10	10		*930*	959	3.1	3	0	1	941	1.2	2	3	1
ft20	20	5		*1165*	1165	0	8	2	1	-	-	-	-	-
abz5	10	10		*1234*	1269	2.8	3	0	1	1238	0.3	3	1	1
abz6	10	10		*943*	952	1.0	7	0	1	948	0.5	2	2	1
abz7	20	15		*656*	742	13.1	3	4	1	729	11.1	3	52	3
abz8	20	15	646	665	742	11.6	8	14	1	723	8.7	3	53	3
abz9	20	15		*678*	774	14.2	4	5	1	744	9.7	2	33	3
yn1	20	20		*884*	1006	13.8	3	6	2	931	5.3	4	98	4
yn2	20	20	870	904	1018	12.6	5	11	2	958	6.0	5	133	5
yn3	20	20	840	892	999	12.0	4	8	2	944	5.8	2	50	3
yn4	20	20	920	968	1114	15.1	3	6	2	1085	12.1	3	83	4
ta01	15	15		*1231*	1387	12.7	5	3	1	1305	6.0	3	22	2
ta02	15	15		*1244*	1375	10.5	3	2	1	1333	7.2	3	21	2
ta03	15	15		*1218*	1357	11.4	5	1	1	1271	4.4	5	35	2
ta04	15	15		*1175*	1263	7.5	4	3	1	1220	3.8	3	19	2
ta05	15	15		*1224*	1344	9.8	4	3	1	1263	3.2	4	27	2
ta06	15	15		*1238*	1410	13.9	3	1	1	1308	5.7	4	28	2
ta07	15	15		*1227*	1318	7.4	6	2	1	1287	4.9	4	24	2
ta08	15	15		*1217*	1309	7.6	5	4	1	1261	3.6	5	33	2
ta09	15	15		*1274*	1353	6.2	7	7	1	1349	5.9	3	21	2
ta10	15	15		*1241*	1395	12.4	4	2	1	1319	6.3	3	22	2

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
ta11	20	15	1323	1357	1588	17.0	4	4	1	1514	11.6	4	66	3
ta12	20	15	1351	1367	1529	11.9	4	6	1	1500	9.7	3	54	3
ta13	20	15	1282	1342	1463	9.0	4	5	1	-	-	1	12	3
ta14	20	15		*1345*	1497	11.3	6	8	1	-	-	1	14	3
ta15	20	15	1304	1339	1529	14.2	3	3	1	1475	10.2	3	48	3
ta16	20	15	1304	1360	1544	13.5	3	4	2	1487	9.3	3	51	3
ta17	20	15		*1462*	1654	13.1	5	7	1	1591	8.8	3	51	3
ta18	20	15	1369	1396	1602	14.8	7	11	1	-	-	1	15	3
ta19	20	15	1304	1332	1518	14.0	8	12	1	1467	10.1	3	53	3
ta20	20	15	1318	1348	1561	15.8	3	4	1	1432	6.2	3	46	3
ta21	20	20	1573	1642	1872	14.0	6	13	2	1810	10.2	2	48	4
ta22	20	20	1542	1600	1806	12.9	4	8	2	1800	12.5	2	48	4
ta23	20	20	1474	1557	1773	13.9	4	7	2	1701	9.2	2	49	4
ta24	20	20	1606	1644	1813	10.3	9	22	2	-	-	1	23	3
ta25	20	20	1518	1595	1800	12.9	5	10	2	1789	12.2	2	50	4
ta26	20	20	1558	1643	1831	11.4	6	14	2	1744	6.1	4	100	4
ta27	20	20	1617	1680	1940	15.5	3	5	2	1845	9.8	6	153	4
ta28	20	20	1591	1603	1758	9.7	4	7	2	-	-	1	23	4
ta29	20	20	1525	1625	1781	9.6	6	14	2	1732	6.6	3	75	4
ta30	20	20	1485	1584	1784	12.6	4	8	2	1710	8.0	4	103	3
ta31	30	15		*1764*	2047	16.0	4	21	2	-	-	1	59	7
ta32	30	15	1774	1784	2150	20.5	5	26	2	-	-	1	64	6
ta33	30	15	1778	1791	2064	15.2	4	21	2	1984	10.8	2	122	6
ta34	30	15	1828	1829	2087	14.1	4	16	2	-	-	1	56	5
ta35	30	15		*2007*	2170	8.1	3	13	2	-	-	1	52	6

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
ta36	30	15		*1819*	2080	14.3	7	41	2	1999	9.9	2	127	6
ta37	30	15		*1771*	2022	14.2	3	16	2	1968	11.1	6	367	6
ta38	30	15		*1673*	1967	17.6	4	21	2	1933	15.5	6	381	6
ta39	30	15		*1795*	2076	15.7	3	14	2	1966	9.5	5	305	6
ta40	30	15	1631	1669	1951	16.9	4	20	2	1936	16.0	3	199	6
ta41	30	20	1876	2005	2441	21.7	6	48	2	2348	17.1	3	288	7
ta42	30	20	1867	1937	2334	20.5	7	57	2	-	-	1	88	6
ta43	30	20	1809	1846	2245	21.6	7	56	2	2152	16.6	2	189	7
ta44	30	20	1927	1979	2438	23.2	5	40	2	2303	16.4	7	629	7
ta45	30	20	1997	2000	2273	13.7	5	39	2	2183	9.2	3	270	7
ta46	30	20	1940	2004	2357	17.6	4	30	2	2342	16.9	2	177	7
ta47	30	20	1789	1889	2234	18.3	6	48	2	-	-	1	91	7
ta48	30	20	1912	1941	2258	16.3	3	22	2	2225	14.6	4	363	7
ta49	30	20	1915	1961	2403	22.5	4	30	2	2338	19.2	2	181	8
ta50	30	20	1807	1923	2373	23.4	4	32	2	2343	21.8	2	188	7
ta51	50	15		*2760*	3025	9.6	4	157	4	-	-	1	477	28
ta52	50	15		*2756*	2993	8.6	5	179	4	2983	8.2	2	875	22
ta53	50	15		*2717*	2995	10.2	3	96	4	2929	7.8	3	1447	24
ta54	50	15		*2839*	2926	3.1	4	137	4	2921	2.9	2	950	18
ta55	50	15		*2679*	3013	12.5	3	89	4	2999	11.9	2	987	21
ta56	50	15		*2781*	2962	6.5	7	256	4	-	-	1	393	19
ta57	50	15		*2943*	3105	5.5	3	83	3	-	-	1	382	20
ta58	50	15		*2885*	3053	5.8	3	97	4	3047	5.6	3	1379	18
ta59	50	15		*2655*	2948	11.0	7	282	4	2926	10.2	3	1339	21
ta60	50	15		*2723*	3003	10.3	7	307	4	2913	7.0	6	3022	24

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
ta61	50	20		*2868*	3228	12.6	5	249	4	3169	10.5	5	3186	26
ta62	50	20		*2869*	3282	14.4	5	250	5	3243	13.0	3	2006	27
ta63	50	20		*2755*	3147	14.2	7	385	5	3127	13.5	3	1999	25
ta64	50	20		*2702*	3126	15.7	3	134	5	3041	12.5	4	2627	24
ta65	50	20		*2725*	3117	14.4	4	213	5	3117	14.4	1	647	25
ta66	50	20		*2845*	3291	15.7	6	320	4	3194	12.3	4	2501	22
ta67	50	20		*2825*	3287	16.4	5	255	4	3166	12.1	5	3100	28
ta68	50	20		*2784*	3182	14.3	5	255	5	3129	12.4	6	3797	23
ta69	50	20		*3071*	3443	12.1	5	242	4	3413	11.1	2	1166	23
ta70	50	20		*2995*	3395	13.4	7	376	5	-	-	1	606	23
ta71	100	20		*5464*	5816	6.4	4	2081	23	-	-	1	6754	180
ta72	100	20		*5181*	5531	6.8	3	1703	33	5418	4.6	5	36328	208
ta73	100	20		*5568*	5951	6.9	4	2101	25	5852	5.1	10	68793	195
ta74	100	20		*5339*	5589	4.7	4	2331	26	5556	4.1	3	20723	193
ta75	100	20		*5392*	5876	9.0	8	5380	26	5802	7.6	4	30631	200
ta76	100	20		*5342*	5780	8.2	4	2089	21	5619	5.2	2	14294	172
ta77	100	20		*5436*	5772	6.2	4	2462	29	5616	3.3	4	29165	206
ta78	100	20		*5394*	5708	5.8	7	4018	22	5660	4.9	3	20521	179
ta79	100	20		*5358*	5583	4.2	9	5757	24	5536	3.3	7	49794	222
ta80	100	20		*5183*	5514	6.4	6	3805	23	5448	5.1	2	14244	151

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
dmu01	20	15	2501	2563	2961	15.5	8	16	2	2709	5.7	3	55	3
dmu02	20	15	2651	2706	3118	15.2	3	4	1	2873	6.2	6	112	3
dmu03	20	15		*2731*	3122	14.3	3	4	1	-	-	1	17	3
dmu04	20	15	2601	2669	2953	10.6	9	16	2	2923	9.5	2	38	3
dmu05	20	15		*2749*	3125	13.7	4	7	1	2994	8.9	5	96	3
dmu06	20	20	2998	3244	3571	10.1	3	6	2	3504	8.0	4	121	4
dmu07	20	20	2815	3046	3427	12.5	6	15	2	3316	8.9	2	55	4
dmu08	20	20	3051	3188	3742	17.4	6	16	2	3475	9.0	2	56	4
dmu09	20	20	2956	3092	3446	11.4	5	12	2	3397	9.9	2	63	5
dmu10	20	20	2858	2984	3259	9.2	6	15	2	3139	5.2	3	84	5
dmu11	30	15	3395	3430	4085	19.1	5	36	2	4081	19.0	2	171	6
dmu12	30	15	3418	3495	4069	16.4	6	45	2	4046	15.8	4	316	6
dmu13	30	15		*3681*	4337	17.8	4	24	2	4166	13.2	3	231	6
dmu14	30	15		*3394*	4013	18.2	4	23	2	3960	16.7	3	222	6
dmu15	30	15		*3343*	3867	15.7	6	40	2	3696	10.6	6	515	7
dmu16	30	20	3734	3751	4421	17.9	4	33	2	4409	17.5	3	346	7
dmu17	30	20	3709	3814	4420	15.9	5	48	2	4405	15.5	2	233	7
dmu18	30	20		*3844*	4741	23.3	6	60	2	4584	19.3	6	689	7
dmu19	30	20	3669	3768	4445	18.0	5	45	2	4360	15.7	3	350	7
dmu20	30	20	3604	3710	4328	16.7	8	78	2	4216	13.6	3	344	8
dmu21	40	15		*4380*	4873	11.3	5	86	3	4758	8.6	5	1136	15
dmu22	40	15		*4725*	5137	8.7	4	62	3	5090	7.7	3	602	10
dmu23	40	15		*4668*	5025	7.6	6	109	3	-	-	1	211	11
dmu24	40	15		*4648*	4938	6.2	5	88	3	4919	5.8	2	410	11
dmu25	40	15		*4164*	4538	9.0	4	67	3	4439	6.6	4	806	11

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
dmu26	40	20		*4647*	5441	17.1	5	135	3	-	-	1	298	13
dmu27	40	20		*4848*	5861	20.9	4	97	3	5624	16.0	8	2391	15
dmu28	40	20		*4692*	5373	14.5	6	143	3	-	-	1	290	12
dmu29	40	20		*4691*	5431	15.8	4	93	3	5400	15.1	2	595	13
dmu30	40	20		*4732*	5595	18.2	3	61	3	5446	15.1	2	603	15
dmu31	50	15		*5640*	5949	5.5	5	186	4	5872	4.1	3	1309	20
dmu32	50	15		*5927*	6062	2.3	4	111	4	5953	0.4	6	2178	21
dmu33	50	15		*5728*	6156	7.5	3	81	4	5879	2.6	6	2354	24
dmu34	50	15		*5385*	5590	3.8	7	256	4	-	-	1	442	19
dmu35	50	15		*5635*	5824	3.4	4	139	4	5777	2.5	2	856	22
dmu36	50	20		*5621*	6567	16.8	7	360	4	6470	15.1	2	1237	21
dmu37	50	20		*5851*	6527	11.6	3	130	4	6465	10.5	4	2571	22
dmu38	50	20		*5713*	6793	18.9	5	252	4	6617	15.8	3	1888	23
dmu39	50	20		*5747*	6610	15.0	10	594	5	6321	10.0	5	3140	23
dmu40	50	20		*5577*	6506	16.7	5	258	4	6405	14.8	3	1863	25
dmu41	20	15	3007	3248	3852	18.6	4	9	1	3698	13.9	4	119	4
dmu42	20	15	3172	3390	3871	14.2	3	6	1	3817	12.6	2	56	4
dmu43	20	15	3292	3441	4054	17.8	4	9	1	3945	14.6	3	84	4
dmu44	20	15	3283	3488	4178	19.8	3	7	1	4021	15.3	5	128	4
dmu45	20	15	3001	3272	3778	15.5	5	11	1	3604	10.1	2	54	4
dmu46	20	20	3575	4035	4754	17.8	5	17	2	4537	12.4	3	134	5
dmu47	20	20	3522	3939	4698	19.3	6	22	2	4276	8.6	5	212	5
dmu48	20	20	3447	3763	4331	15.1	3	8	2	4158	10.5	5	200	5
dmu49	20	20	3403	3710	4405	18.7	6	26	2	4375	17.9	2	86	4
dmu50	20	20	3496	3729	4401	18.0	5	16	2	4283	14.9	5	197	5

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
dmu51	30	15	3917	4167	5066	21.6	3	30	2	4995	19.9	3	406	9
dmu52	30	15	4065	4311	5272	22.3	4	46	2	5011	16.2	2	309	11
dmu53	30	15	4141	4394	5450	24.0	3	22	2	5232	19.1	2	210	7
dmu54	30	15	4202	4362	5221	19.7	7	76	2	5156	18.2	2	239	9
dmu55	30	15	4140	4271	5178	21.2	7	59	2	-	-	1	101	7
dmu56	30	20	4554	4941	5918	19.8	4	61	2	5783	17.0	4	746	10
dmu57	30	20	4302	4655	5747	23.5	3	35	2	5585	20.0	3	503	10
dmu58	30	20	4319	4708	5746	22.0	4	51	2	5502	16.9	5	814	10
dmu59	30	20	4217	4624	5782	25.0	5	73	2	5643	22.0	2	329	11
dmu60	30	20	4319	4755	5682	19.5	3	36	2	-	-	1	177	11
dmu61	40	15	4917	5172	6571	27.0	3	67	4	6322	22.2	4	1192	21
dmu62	40	15	5033	5265	6524	23.9	5	133	4	6198	17.7	3	897	18
dmu63	40	15	5111	5326	6461	21.3	7	192	4	6343	19.1	4	1231	20
dmu64	40	15	5130	5250	6757	28.7	4	91	3	6295	19.9	4	1181	19
dmu65	40	15	5105	5190	6376	22.9	4	92	3	6199	19.4	4	1067	18
dmu66	40	20	5391	5717	7385	29.2	4	150	4	7135	24.8	4	2060	26
dmu67	40	20	5589	5813	7149	23.0	7	268	3	6952	19.6	4	1793	18
dmu68	40	20	5426	5773	7402	28.2	7	313	4	6820	18.1	4	2063	22
dmu69	40	20	5423	5709	7141	25.1	3	82	3	6933	21.4	3	1195	16
dmu70	40	20	5501	5889	7687	30.5	3	97	4	7214	22.5	3	1380	21
dmu71	50	15	6080	6223	7685	23.5	5	294	6	7634	22.7	2	1359	39
dmu72	50	15	6395	6483	7981	23.1	4	189	6	7543	16.4	2	1255	34
dmu73	50	15	6001	6163	7547	22.5	5	259	5	7211	17.0	5	2795	36
dmu74	50	15	6123	6220	7790	25.2	8	397	5	7538	21.2	5	3343	39
dmu75	50	15	6029	6197	7614	22.9	6	273	5	7388	19.2	5	2644	33

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	LB	UB	H = 10					H = 100				
					Best	Gap[c]	# I[d]	CPU[e]	Mem[f]	Best	Gap[c]	# I[d]	CPU[e]	Mem[f]
dmu76	50	20	6342	6813	8511	24.9	5	433	6	8126	19.3	4	3967	42
dmu77	50	20	6499	6822	8637	26.6	3	160	5	8433	23.6	2	1897	37
dmu78	50	20	6586	6770	8364	23.5	9	785	6	8213	21.3	3	3250	40
dmu79	50	20	6650	6970	8797	26.2	6	498	7	8727	25.2	3	2883	47
dmu80	50	20	6459	6686	8184	22.4	4	300	6	-	-	1	990	46
swv01	20	10		*1407*	1579	12.2	6	7	1	1503	6.8	3	35	3
swv02	20	10		*1475*	1589	7.7	3	2	1	-	-	1	11	3
swv03	20	10		*1398*	1581	13.1	4	3	1	1517	8.5	3	34	3
swv04	20	10	1450	1467	1731	18.0	4	6	1	-	-	1	13	3
swv05	20	10		*1424*	1648	15.7	4	3	1	1584	11.2	4	50	3
swv06	20	15	1591	1671	1961	17.4	4	8	1	1918	14.8	2	41	4
swv07	20	15	1447	1594	1762	10.5	3	4	1	-	-	1	24	4
swv08	20	15	1641	1752	2077	18.6	4	8	1	1993	13.8	3	65	4
swv09	20	15	1605	1655	1995	20.5	6	13	1	1992	20.4	2	44	4
swv10	20	15	1632	1743	1999	14.7	3	5	1	1976	13.4	2	43	4
swv11	50	10		*2983*	3427	14.9	6	191	4	3345	12.1	4	1169	20
swv12	50	10	2972	2977	3487	17.1	4	117	5	3324	11.7	3	1103	31
swv13	50	10		*3104*	3521	13.4	3	80	4	3369	8.5	2	769	25
swv14	50	10		*2968*	3276	10.4	4	110	4	3209	8.1	4	1385	27
swv15	50	10		*2885*	3468	20.2	3	67	4	3306	14.6	2	676	25
swv16	50	10		*2924*	2924	0	5	83	3	-	-	-	-	-
swv17	50	10		*2794*	2794	0	7	133	3	-	-	-	-	-
swv18	50	10		*2852*	2852	0	4	63	3	-	-	-	-	-
swv19	50	10		*2843*	2884	1.4	3	44	3	2843	0	3	634	16
swv20	50	10		*2823*	2827	0.1	3	43	3	2823	0	2	403	12

[a] # Jobs
[b] # Machines
[c] Relative gap of Best with UB (%)
[d] # Iterations
[e] Sum of CPU over all iterations (s)
[f] Max Memory used over all iterations (MB)

Table A.2: *Results from iteratively finding JSSP solutions by DP (continued)*

Instance	# J[a]	# M[b]	CPU[c]	Mem[d]
ft06	6	6	0	0
ft10	10	10	28	13
ft20	20	5	0	0
dmu13	30	15	0	1
dmu14	30	15	0	1
dmu15	30	15	0	1
dmu18	30	20	0	2
dmu21	40	15	1	1
dmu22	40	15	0	1
dmu23	40	15	0	1
dmu24	40	15	0	1
dmu25	40	15	0	1
dmu26	40	20	0	1
dmu27	40	20	0	1
dmu28	40	20	0	1
dmu29	40	20	0	1
dmu30	40	20	0	1
dmu31	50	15	0	1
dmu32	50	15	0	1
dmu33	50	15	0	1
dmu34	50	15	0	1
dmu35	50	15	0	1
dmu36	50	20	0	2
dmu37	50	20	0	2
dmu38	50	20	0	2
dmu39	50	20	0	2
dmu40	50	20	1	2
la01	10	5	0	0
la02	10	5	0	0
la03	10	5	0	0
la04	10	5	0	0
la05	10	5	0	0
la06	15	5	0	0
la07	15	5	0	0
la08	15	5	0	0
la09	15	5	0	0
la10	15	5	0	0
la11	20	5	0	0
la12	20	5	0	0
la13	20	5	0	0
la14	20	5	0	0
la15	20	5	0	0
la16	10	10	11	8
la17	10	10	0	1
la18	10	10	5	3
la19	10	10	3	2
la20	10	10	1	1
la22	15	10	105	34
la23	15	10	0	0
la24	15	10	100	24

Instance	# J[a]	# M[b]	CPU[c]	Mem[d]
la26	20	10	0	1
la27	20	10	0	1
la28	20	10	0	0
la30	20	10	0	1
la31	30	10	0	1
la32	30	10	0	1
la33	30	10	0	1
la34	30	10	0	1
la35	30	10	0	1
la37	15	15	0	1
la39	15	15	36	12
abz5	10	10	14	8
abz6	10	10	1	1
orb01	10	10	16	7
orb02	10	10	10	5
orb03	10	10	87	24
orb04	10	10	7	5
orb05	10	10	11	4
orb06	10	10	17	6
orb07	10	10	3	2
orb08	10	10	0	0
orb09	10	10	5	3
orb10	10	10	0	1
swv02	20	10	0	1
swv11	50	10	0	1
swv13	50	10	0	1
swv14	50	10	0	1
swv15	50	10	0	1
swv16	50	10	0	1
swv17	50	10	0	1
swv18	50	10	0	1
swv19	50	10	0	1
swv20	50	10	0	1
ta01	15	15	1024	304
ta14	20	15	0	1
ta17	20	15	2763	457
ta31	30	15	0	1
ta36	30	15	0	1
ta37	30	15	0	1
ta38	30	15	0	1
ta39	30	15	0	1
ta51	50	15	0	1
ta52	50	15	0	1
ta53	50	15	0	1
ta54	50	15	0	1
ta55	50	15	0	1
ta56	50	15	0	1
ta57	50	15	0	1
ta58	50	15	0	1
ta59	50	15	0	1
ta60	50	15	0	1

[a] # Jobs [b] # Machines [c] CPU (s) [d] Memory (MB)

Table A.3: *Optimality proven by finding lower bound with DP*

Instance	# J[a]	# M[b]	CPU[c]	Mem[d]
ta61	50	20	0	2
ta62	50	20	0	2
ta63	50	20	0	2
ta64	50	20	0	2
ta65	50	20	0	2
ta66	50	20	0	2
ta67	50	20	1	2
ta68	50	20	0	2
ta69	50	20	0	2
ta70	50	20	0	2
ta71	100	20	0	3
ta72	100	20	0	3
ta73	100	20	0	3
ta74	100	20	0	3
ta75	100	20	0	3
ta76	100	20	0	3
ta77	100	20	0	3
ta78	100	20	0	3
ta79	100	20	0	3
ta80	100	20	1	3

[a] # Jobs [b] # Machines [c] CPU (s) [d] Memory (MB)

Table A.3: *Optimality proven by finding lower bound with DP (continued)*

Instance	# J[a]	# M[b]	CPU[c]	Mem[d]
abz7	20	15	262	649
abz9	20	15	260	543
la21	15	10	199	256
la25	15	10	236	267
la29	20	10	118	288
la36	15	15	144	318
la38	15	15	208	396
la40	15	15	501	347
swv01	20	10	556	365
swv03	20	10	368	516
swv05	20	10	571	350
yn1	20	20	380	760
dmu03	20	15	366	642
dmu05	20	15	481	638
ta02	15	15	336	370
ta03	15	15	232	382
ta04	15	15	233	378
ta05	15	15	234	381
ta06	15	15	171	334
ta07	15	15	168	313
ta08	15	15	193	333
ta09	15	15	147	337
ta10	15	15	209	400
ta35	30	15	367	1136

[a] # Jobs [b] # Machines [c] CPU (s) [d] Memory (MB)

Table A.4: *Optimality not proven by finding lower bound with DP*

A

Instance	Best[a b]		# Iterations[a]		CPU[a c]		Memory[a d]	
$\mathrm{ft06 \vert h \vert max \vert 1 \vert h \vert max \vert 1}$	103	**98**	3	2	13	34	3	12
$\mathrm{ft06 \vert h \vert max \vert 1 \vert h \vert max \vert}\frac{1}{3}$	**71**		3		7		3	
$\mathrm{ft06 \vert h \vert max \vert 1 \vert h \vert sum \vert}\frac{1}{5}$	98	**94**	3	2	12	29	3	12
$\mathrm{ft06 \vert h \vert max \vert 1 \vert nh \vert max \vert 1}$	*97*	✓	3	1	8	3	3	2
$\mathrm{ft06 \vert h \vert max \vert 1 \vert nh \vert max \vert}\frac{1}{3}$	**71**		3		8		3	
$\mathrm{ft06 \vert h \vert max \vert 1 \vert nh \vert sum \vert}\frac{1}{5}$	**88**		3		8		3	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert h \vert max \vert 1}$	**79**		3		4		2	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert h \vert max \vert}\frac{1}{3}$	**64**		3		6		3	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert h \vert sum \vert}\frac{1}{5}$	**76**		3		5		3	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert nh \vert max \vert 1}$	**79**		3		2		3	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert nh \vert max \vert}\frac{1}{3}$	**64**		3		6		3	
$\mathrm{ft06 \vert h \vert max \vert}\frac{3}{2}\mathrm{\vert nh \vert sum \vert}\frac{1}{5}$	**76**		3		5		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert h \vert max \vert 1}$	**79**		3		4		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert h \vert max \vert}\frac{1}{3}$	**64**		3		5		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert h \vert sum \vert}\frac{1}{5}$	**76**		3		5		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert nh \vert max \vert 1}$	**79**		3		3		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert nh \vert max \vert}\frac{1}{3}$	**64**		3		4		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{1}{3}\mathrm{\vert nh \vert sum \vert}\frac{1}{5}$	**76**		3		5		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert h \vert max \vert 1}$	**65**		2		1		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert h \vert max \vert}\frac{1}{3}$	**59**		2		0		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert h \vert sum \vert}\frac{1}{5}$	**64**		2		1		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert nh \vert max \vert 1}$	**65**		2		0		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert nh \vert max \vert}\frac{1}{3}$	**59**		2		1		3	
$\mathrm{ft06 \vert h \vert sum \vert}\frac{2}{3}\mathrm{\vert nh \vert sum \vert}\frac{1}{5}$	**63**		2		0		3	
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert h \vert max \vert 1}$	106	**100**	3	2	14	40	3	11
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert h \vert max \vert}\frac{1}{3}$	**71**		3		7		3	
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert h \vert sum \vert}\frac{1}{5}$	100	**95**	3	2	14	32	3	11
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert nh \vert max \vert 1}$	104	**99**	3	2	13	39	3	12
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert nh \vert max \vert}\frac{1}{3}$	**71**		3		7		3	
$\mathrm{ft06 \vert nh \vert max \vert 1 \vert nh \vert sum \vert}\frac{1}{5}$	**89**		3		8		3	

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances*

Instance	Best[ab]			# Iterations[a]			CPU[ac]			Memory[ad]		
ft06|nh|max|$\frac{3}{2}$|h|max|1	**79**			3			5			2		
ft06|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	**64**			3			6			3		
ft06|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**76**			3			5			3		
ft06|nh|max|$\frac{3}{2}$|nh|max|1	**79**			3			5			3		
ft06|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	**64**			3			4			3		
ft06|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	**76**			3			5			3		
ft06|nh|sum|$\frac{1}{3}$|h|max|1	91	**88**		3	2		9	29		3	10	
ft06|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	**67**			3			6			3		
ft06|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	87	**86**		3	1		10	17		3	8	
ft06|nh|sum|$\frac{1}{3}$|nh|max|1	89	**86**		3	2		9	24		3	10	
ft06|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	**66**			3			8			3		
ft06|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	**79**			3			8			3		
ft06|nh|sum|$\frac{2}{3}$|h|max|1	**66**			3			1			3		
ft06|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	**60**			3			4			3		
ft06|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	**65**			3			1			3		
ft06|nh|sum|$\frac{2}{3}$|nh|max|1	**66**			3			2			3		
ft06|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	**59**			3			4			3		
ft06|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	**64**			3			1			3		
la01|h|max|1|h|max|1	1356	**1352**		4	2		68	162		5	19	
la01|h|max|1|h|max|$\frac{1}{3}$	906	901	**897**	3	2	2	53	357	1477	4	20	161
la01|h|max|1|h|sum|$\frac{1}{5}$	1621	1609	**1604**	3	2	2	51	370	1595	5	27	208
la01|h|max|1|nh|max|1	1367	**1352**		4	2		72	194		4	25	
la01|h|max|1|nh|max|$\frac{1}{3}$	904	901	**897**	3	2	2	53	369	1554	4	20	164
la01|h|max|1|nh|sum|$\frac{1}{5}$	**1604**			3			32			4		
la01|h|max|$\frac{3}{2}$|h|max|1	**1254**			4			44			4		
la01|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	**864**			3			23			4		
la01|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**1470**			3			25			4		
la01|h|max|$\frac{3}{2}$|nh|max|1	**1254**			3			23			4		
la01|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	**864**			3			25			4		
la01|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	1484	**1470**		3	2		42	169		4	27	

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[ab]				# Iterations[a]				CPU[ac]				Memory[ad]			
la01|h|sum|$\frac{1}{3}$|h|max|1	1245	1232	1179	999	2	3	2	2	13	329	1878	15728	4	29	234	2050
la01|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	907	868	795	765	3	2	2	2	30	229	1802	11984	4	28	230	1793
la01|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1386	1366	1345	1107	3	3	2	2	29	351	1875	16961	4	35	242	2050
la01|h|sum|$\frac{1}{3}$|nh|max|1	1234	1216	1173	1155	2	3	2	2	12	323	1834	14952	4	28	235	2012
la01|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	811	782	765	-	3	2	2	1	26	202	1160	3100	4	26	214	1445
la01|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1363	-	1310	1285	2	1	3	2	13	106	2841	16109	4	29	243	2007
la01|h|sum|$\frac{2}{3}$|h|max|1	**764**				4				17				4			
la01|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	700	**699**			4	2			25	60			4	20		
la01|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	821	**800**			4	2			24	60			4	23		
la01|h|sum|$\frac{2}{3}$|nh|max|1	777	**764**			4	2			24	58			4	23		
la01|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	**699**				4				19				4			
la01|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	822	**800**			3	2			15	59			4	24		
la01|nh|max|1|h|max|1	1356	**1352**			4	2			72	156			4	19		
la01|nh|max|1|h|max|$\frac{1}{3}$	911	901	**897**		4	2	2		77	367	1443		4	21	158	
la01|nh|max|1|h|sum|$\frac{1}{5}$	1608	**1604**			4	2			72	175			5	23		
la01|nh|max|1|nh|max|1	1367	**1352**			3	2			55	191			4	24		
la01|nh|max|1|nh|max|$\frac{1}{3}$	906	898	**897**		3	2	2		56	368	1473		4	20	158	
la01|nh|max|1|nh|sum|$\frac{1}{5}$	1606	**1604**			3	2			56	193			4	23		
la01|nh|max|$\frac{3}{2}$|h|max|1	1583	**1254**			2	2			24	211			4	33		
la01|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	**864**				5				66				4			
la01|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**1470**				4				45				4			
la01|nh|max|$\frac{3}{2}$|nh|max|1	**1254**				4				45				4			
la01|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	**864**				4				45				4			
la01|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	**1470**				4				47				4			
la01|nh|sum|$\frac{1}{3}$|h|max|1	1151	-	-	1014	2	1	1	2	14	101	882	17155	4	29	239	2071
la01|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	886	765	-	-	3	2	1	1	28	224	496	2695	4	28	177	1252
la01|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1438	-	1068	-	5	1	2	1	66	125	1953	4680	5	36	294	1606
la01|nh|sum|$\frac{1}{3}$|nh|max|1	1233	-	1199	999	3	1	2	2	28	104	1874	16121	4	28	230	2052
la01|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	908	832	816	765	2	3	2	2	15	390	1883	14570	4	28	237	1897
la01|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1448	-	1388	1107	2	1	2	2	15	144	2364	18422	4	33	279	2570

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[a b]				# Iterations[a]				CPU[a c]				Memory[a d]			
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{max}\vert 1$	**764**				3				11				4			
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{1}{3}$	700	**699**			2	2			9	60			4	20		
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{5}$	**800**				3				11				5			
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert 1$	**764**				4				18				4			
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert\frac{1}{3}$	703	**699**			3	2			15	63			4	21		
$\mathrm{la01}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{1}{5}$	**800**				3				9				4			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{h}\vert\mathrm{max}\vert 1$	1317	**1249**			3	2			38	159			5	22		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{h}\vert\mathrm{max}\vert\frac{1}{3}$	871	**853**			4	2			69	142			5	18		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{5}$	1570	**1417**			4	2			67	219			4	31		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{nh}\vert\mathrm{max}\vert 1$	1336	**1249**			3	2			42	191			5	25		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{nh}\vert\mathrm{max}\vert\frac{1}{3}$	899	**853**			3	2			44	197			4	23		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert 1\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{1}{5}$	**1417**				4				53				4			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{h}\vert\mathrm{max}\vert 1$	**1051**				3				18				4			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{1}{3}$	**787**				4				33				4			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{5}$	1168	**1163**			3	2			29	94			4	21		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{nh}\vert\mathrm{max}\vert 1$	1078	**1051**			4	2			46	117			4	23		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{nh}\vert\mathrm{max}\vert\frac{1}{3}$	*787*	✓			3	1			20	2			4	2		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{3}{2}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{1}{5}$	1189	**1163**			3	2			32	120			4	24		
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{h}\vert\mathrm{max}\vert 1$	1339	1330	1227	972	2	2	2	2	16	253	2198	17828	4	29	238	2004
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{1}{3}$	754	-	-	-	3	1	1	1	24	67	524	3627	5	23	194	1497
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{5}$	1515	1479	1357	1056	4	2	2	2	52	300	2491	19031	5	36	289	1999
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert 1$	1212	-	-	972	3	1	1	2	32	120	1057	18004	5	29	237	2006
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert\frac{1}{3}$	847	-	754	-	3	1	2	1	30	113	1848	3810	4	29	236	1510
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{3}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{1}{5}$	1530	1400	-	1307	2	3	1	2	17	427	1243	21022	4	34	285	2430
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{max}\vert 1$	**754**				3				12				5			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{max}\vert\frac{1}{3}$	**688**				3				11				5			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{1}{5}$	*782*	-	✓		4	1	1		22	10	10		5	13	12	
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert 1$	**754**				3				11				5			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{max}\vert\frac{1}{3}$	**688**				3				12				5			
$\mathrm{la02}\vert\mathrm{h}\vert\mathrm{sum}\vert\frac{2}{3}\vert\mathrm{nh}\vert\mathrm{sum}\vert\frac{1}{5}$	886	*782*	✓		4	2	1		26	87	12		5	27	16	

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold, solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

<table>
<tr><th>Instance</th><th colspan="4">Best^{ab}</th><th colspan="4"># Iterations^{a}</th><th colspan="4">CPU^{ac}</th><th colspan="4">Memory^{ad}</th></tr>
<tr><td>la02|nh|max|1|h|max|1</td><td>1302</td><td>-</td><td>1260</td><td></td><td>3</td><td>1</td><td>2</td><td></td><td>39</td><td>135</td><td>1167</td><td></td><td>5</td><td>20</td><td>151</td><td></td></tr>
<tr><td>la02|nh|max|1|h|max|$\frac{1}{3}$</td><td>914</td><td>853</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>50</td><td>219</td><td></td><td></td><td>5</td><td>24</td><td></td><td></td></tr>
<tr><td>la02|nh|max|1|h|sum|$\frac{1}{5}$</td><td>1509</td><td>-</td><td>1428</td><td></td><td>4</td><td>1</td><td>2</td><td></td><td>67</td><td>173</td><td>1580</td><td></td><td>5</td><td>26</td><td>206</td><td></td></tr>
<tr><td>la02|nh|max|1|nh|max|1</td><td>1324</td><td>1249</td><td></td><td></td><td>4</td><td>2</td><td></td><td></td><td>69</td><td>173</td><td></td><td></td><td>4</td><td>22</td><td></td><td></td></tr>
<tr><td>la02|nh|max|1|nh|max|$\frac{1}{3}$</td><td>1059</td><td>853</td><td></td><td></td><td>2</td><td>3</td><td></td><td></td><td>24</td><td>441</td><td></td><td></td><td>4</td><td>29</td><td></td><td></td></tr>
<tr><td>la02|nh|max|1|nh|sum|$\frac{1}{5}$</td><td>1420</td><td>1417</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>38</td><td>123</td><td></td><td></td><td>4</td><td>20</td><td></td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|h|max|1</td><td>1076</td><td>1051</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>34</td><td>110</td><td></td><td></td><td>4</td><td>22</td><td></td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$</td><td>804</td><td>795</td><td>787</td><td></td><td>4</td><td>2</td><td>2</td><td></td><td>46</td><td>224</td><td>966</td><td></td><td>4</td><td>22</td><td>161</td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$</td><td>1200</td><td>-</td><td>1163</td><td></td><td>4</td><td>1</td><td>2</td><td></td><td>50</td><td>116</td><td>1073</td><td></td><td>4</td><td>23</td><td>197</td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|nh|max|1</td><td>1077</td><td>1051</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>33</td><td>114</td><td></td><td></td><td>4</td><td>23</td><td></td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$</td><td>792</td><td>-</td><td>787</td><td></td><td>4</td><td>1</td><td>2</td><td></td><td>50</td><td>108</td><td>982</td><td></td><td>4</td><td>21</td><td>161</td><td></td></tr>
<tr><td>la02|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$</td><td>1163</td><td></td><td></td><td></td><td>3</td><td></td><td></td><td></td><td>21</td><td></td><td></td><td></td><td>4</td><td></td><td></td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|h|max|1</td><td>1303</td><td>-</td><td>1235</td><td>988</td><td>3</td><td>1</td><td>3</td><td>2</td><td>33</td><td>128</td><td>3637</td><td>18704</td><td>4</td><td>28</td><td>239</td><td>2021</td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$</td><td>823</td><td>-</td><td>754</td><td>-</td><td>3</td><td>1</td><td>2</td><td>1</td><td>32</td><td>116</td><td>1751</td><td>3719</td><td>4</td><td>28</td><td>231</td><td>1358</td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$</td><td>1235</td><td>-</td><td>1056</td><td>-</td><td>4</td><td>1</td><td>2</td><td>1</td><td>49</td><td>116</td><td>2004</td><td>6646</td><td>5</td><td>28</td><td>232</td><td>1684</td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|nh|max|1</td><td>1385</td><td>-</td><td>1213</td><td>988</td><td>2</td><td>1</td><td>2</td><td>2</td><td>18</td><td>148</td><td>2614</td><td>18568</td><td>5</td><td>33</td><td>286</td><td>2012</td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$</td><td>888</td><td>754</td><td>-</td><td>-</td><td>3</td><td>2</td><td>1</td><td>1</td><td>33</td><td>238</td><td>603</td><td>3957</td><td>4</td><td>29</td><td>183</td><td>1383</td></tr>
<tr><td>la02|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$</td><td>1371</td><td>-</td><td>1036</td><td>-</td><td>3</td><td>1</td><td>2</td><td>1</td><td>34</td><td>138</td><td>2090</td><td>6108</td><td>4</td><td>33</td><td>283</td><td>1623</td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|h|max|1</td><td>838</td><td>754</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>17</td><td>73</td><td></td><td></td><td>5</td><td>27</td><td></td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$</td><td>688</td><td></td><td></td><td></td><td>4</td><td></td><td></td><td></td><td>19</td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$</td><td>834</td><td>782</td><td>✓</td><td></td><td>4</td><td>2</td><td>1</td><td></td><td>23</td><td>73</td><td>10</td><td></td><td>4</td><td>25</td><td>12</td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|nh|max|1</td><td>848</td><td>754</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>17</td><td>75</td><td></td><td></td><td>4</td><td>27</td><td></td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$</td><td>688</td><td></td><td></td><td></td><td>4</td><td></td><td></td><td></td><td>19</td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr>
<tr><td>la02|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$</td><td>782</td><td>-</td><td>✓</td><td></td><td>3</td><td>1</td><td>1</td><td></td><td>13</td><td>11</td><td>12</td><td></td><td>4</td><td>15</td><td>16</td><td></td></tr>
<tr><td>la03|h|max|1|h|max|1</td><td>1158</td><td></td><td></td><td></td><td>3</td><td></td><td></td><td></td><td>26</td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr>
<tr><td>la03|h|max|1|h|max|$\frac{1}{3}$</td><td>910</td><td>811</td><td>800</td><td>✓</td><td>3</td><td>3</td><td>2</td><td>1</td><td>41</td><td>546</td><td>2166</td><td>3151</td><td>4</td><td>28</td><td>158</td><td>745</td></tr>
<tr><td>la03|h|max|1|h|sum|$\frac{1}{5}$</td><td>1320</td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td><td>72</td><td></td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr>
<tr><td>la03|h|max|1|nh|max|1</td><td>1146</td><td></td><td></td><td></td><td>4</td><td></td><td></td><td></td><td>42</td><td></td><td></td><td></td><td>4</td><td></td><td></td><td></td></tr>
<tr><td>la03|h|max|1|nh|max|$\frac{1}{3}$</td><td>814</td><td>798</td><td>-</td><td>-</td><td>3</td><td>2</td><td>1</td><td>1</td><td>41</td><td>297</td><td>956</td><td>5062</td><td>4</td><td>21</td><td>152</td><td>1257</td></tr>
<tr><td>la03|h|max|1|nh|sum|$\frac{1}{5}$</td><td>1446</td><td>1320</td><td></td><td></td><td>3</td><td>2</td><td></td><td></td><td>37</td><td>233</td><td></td><td></td><td>4</td><td>33</td><td></td><td></td></tr>
</table>

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[a b]				# Iterations[a]				CPU[a c]				Memory[a d]			
la03|h|max|$\frac{3}{2}$|h|max|1	982	**952**			3	2			29	126			4	22		
la03|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	755	723	-	**721**	4	2	1	2	51	255	766	6596	4	24	149	1319
la03|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**1060**				3				19				4			
la03|h|max|$\frac{3}{2}$|nh|max|1	**944**				4				31				4			
la03|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	753	722	-	**717**	4	2	1	2	54	262	811	6950	4	25	151	1344
la03|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	1113	**1060**			3	2			35	141			5	23		
la03|h|sum|$\frac{1}{3}$|h|max|1	1085	-	951	882	4	1	2	2	42	112	2001	15983	4	29	218	1923
la03|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	761	-	695	**681**	5	1	3	2	55	113	2232	7098	5	29	233	1434
la03|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1256	1229	1221	963	4	2	2	2	48	254	2225	17169	5	29	223	2024
la03|h|sum|$\frac{1}{3}$|nh|max|1	1199	1126	1097	876	3	2	2	2	29	255	2190	16436	4	28	215	1936
la03|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	747	-	692	**678**	6	1	2	2	74	101	1834	7110	5	27	224	1422
la03|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1372	1233	1196	963	2	2	2	2	15	263	2301	17159	4	33	230	2060
la03|h|sum|$\frac{2}{3}$|h|max|1	701	692	-	**688**	4	2	1	1	23	115	362	698	4	19	141	343
la03|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	632	**628**			3	2			17	54			5	15		
la03|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	784	719	-	**715**	3	3	1	1	16	188	377	1124	4	26	150	514
la03|h|sum|$\frac{2}{3}$|nh|max|1	693	692	-	**686**	3	2	1	1	15	109	379	756	5	19	143	383
la03|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	632	**627**			3	2			18	53			6	16		
la03|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	793	**706**			3	2			16	80			4	28		
la03|nh|max|1|h|max|1	**1249**				4				50				5			
la03|nh|max|1|h|max|$\frac{1}{3}$	846	**829**			4	2			71	179			4	20		
la03|nh|max|1|h|sum|$\frac{1}{5}$	1441	**1438**			3	2			42	135			5	23		
la03|nh|max|1|nh|max|1	**1235**				3				29				4			
la03|nh|max|1|nh|max|$\frac{1}{3}$	825	**822**			7	2			125	130			4	18		
la03|nh|max|1|nh|sum|$\frac{1}{5}$	1564	**1438**			3	2			40	252			4	32		
la03|nh|max|$\frac{3}{2}$|h|max|1	*953*	✓			4	1			36	2			4	2		
la03|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	*723*	-	-	✓	5	1	1	1	69	90	510	674	5	18	139	201
la03|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	*1061*	✓			4	1			35	2			5	2		
la03|nh|max|$\frac{3}{2}$|nh|max|1	*945*	✓			3	1			22	2			4	2		
la03|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	733	*722*	-	✓	4	2	1	1	57	221	573	921	4	20	142	269
la03|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	1103	**1061**			5	2			65	147			4	22		

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[a b]				# Iterations[a]				CPU[a c]				Memory[a d]			
la03|nh|sum|$\frac{1}{3}$|h|max|1	1115	941	893	882	2	2	2	2	16	259	2088	14298	4	29	219	1567
la03|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	745	705	702	**681**	3	3	2	2	36	301	1582	7058	5	28	171	1376
la03|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1279	1081	987	974	2	2	3	2	16	254	3223	15713	5	29	219	1675
la03|nh|sum|$\frac{1}{3}$|nh|max|1	1065	935	915	887	3	2	2	2	29	255	2096	16407	4	26	210	1731
la03|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	796	700	699	**678**	3	2	2	2	37	254	1363	7137	5	31	169	1378
la03|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1300	1022	-	963	4	2	1	2	54	266	1049	18689	5	29	212	1884
la03|nh|sum|$\frac{2}{3}$|h|max|1	725	692	-	**688**	3	2	1	1	15	108	297	811	4	22	150	463
la03|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	632	**628**			3	2			19	60			6	15		
la03|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	758	719	-	**715**	3	2	1	1	16	104	296	1137	4	24	157	744
la03|nh|sum|$\frac{2}{3}$|nh|max|1	693	692	-	**686**	3	2	1	1	15	91	316	961	4	20	153	533
la03|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	632	**627**			4	2			27	60			5	15		
la03|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	784	**706**			3	2			15	76			4	26		
la04|h|max|1|h|max|1	1178	**1126**			3	2			40	146			5	21		
la04|h|max|1|h|max|$\frac{1}{3}$	785	*753*	✓		3	2	1		42	238	108		4	22	26	
la04|h|max|1|h|sum|$\frac{1}{5}$	1197	**1186**			5	2			71	115			4	19		
la04|h|max|1|nh|max|1	1137	*1078*	✓		5	2	1		73	228	240		4	24	89	
la04|h|max|1|nh|max|$\frac{1}{3}$	773	*740*	✓		3	3	1		45	300	98		4	23	26	
la04|h|max|1|nh|sum|$\frac{1}{5}$	1194	**1186**			5	2			75	115			4	19		
la04|h|max|$\frac{3}{2}$|h|max|1	993	972	**959**		4	2	2		45	196	665		4	21	152	
la04|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	733	721	-	**712**	4	3	1	2	52	338	692	3977	4	22	149	1237
la04|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	1009	-	**999**		3	1	3		29	83	1149		4	19	150	
la04|h|max|$\frac{3}{2}$|nh|max|1	978	**959**			4	2			43	106			4	20		
la04|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	733	728	713	**707**	3	2	3	1	38	259	2250	1909	5	23	167	763
la04|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	1019	**999**			4	2			46	107			4	20		
la04|h|sum|$\frac{1}{3}$|h|max|1	1299	878	-	**861**	4	2	1	2	41	196	690	8061	4	26	175	1416
la04|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	684	680	**666**		3	2	3		27	163	1228		4	22	137	
la04|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1253	908	-	**891**	2	2	1	2	13	189	712	8505	4	26	179	1426
la04|h|sum|$\frac{1}{3}$|nh|max|1	1314	1249	861	-	3	2	4	1	29	229	2782	2460	4	26	218	1192
la04|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	717	678	**666**		3	2	2		34	172	833		5	25	141	
la04|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1293	954	940	891	2	2	2	2	12	221	1788	10279	4	26	200	1600

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[ab]				# Iterations[a]				CPU[ac]				Memory[ad]			
la04|h|sum|$\frac{2}{3}$|h|max|1	708	693	**673**		3	2	2		13	94	371		4	20	152	
la04|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	635	**608**			3	3			20	115			5	21		
la04|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	710	703	**683**		4	2	2		18	95	388		4	20	153	
la04|h|sum|$\frac{2}{3}$|nh|max|1	705	685	**667**		3	2	2		16	95	325		4	21	149	
la04|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	632	**608**			3	3			20	115			5	20		
la04|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	710	701	**680**		3	3	2		15	143	409		4	21	151	
la04|nh|max|1|h|max|1	1150	**1126**			4	2			56	129			4	20		
la04|nh|max|1|h|max|$\frac{1}{3}$	769	767	**756**		4	2	2		58	244	951		4	17	139	
la04|nh|max|1|h|sum|$\frac{1}{5}$	1199	**1186**			6	2			85	126			5	20		
la04|nh|max|1|nh|max|1	1107	-	**1080**		4	1	2		55	127	1240		4	21	159	
la04|nh|max|1|nh|max|$\frac{1}{3}$	782	*753*	-	✓	4	3	1	1	62	402	649	892	4	22	139	235
la04|nh|max|1|nh|sum|$\frac{1}{5}$	**1186**				4				44				4			
la04|nh|max|$\frac{3}{2}$|h|max|1	1013	993	*972*	✓	4	2	2	1	47	228	1249	607	4	21	154	239
la04|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	733	-	-	**727**	3	1	1	2	36	123	862	5849	4	21	148	1303
la04|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	1054	1033	*1012*	✓	4	2	2	1	47	228	1231	624	4	21	154	251
la04|nh|max|$\frac{3}{2}$|nh|max|1	978	-	**959**		5	1	2		56	96	721		4	19	150	
la04|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	728	-	-	**716**	3	1	1	2	35	121	875	5590	4	21	154	1315
la04|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	1027	1003	**999**		4	3	2		44	248	448		4	19	139	
la04|nh|sum|$\frac{1}{3}$|h|max|1	1139	910	892	**861**	2	4	2	2	13	465	1634	7743	4	27	195	1435
la04|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	693	684	*671*	✓	3	2	2	1	30	167	1223	474	5	24	144	110
la04|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1349	908	-	**891**	4	2	1	2	44	197	724	7098	4	26	174	1333
la04|nh|sum|$\frac{1}{3}$|nh|max|1	1280	1275	873	**861**	3	2	2	2	31	233	1717	7741	4	27	222	1345
la04|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	691	-	**666**		4	1	4		43	88	2300		4	24	162	
la04|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1188	908	-	**891**	2	2	1	2	14	197	764	8379	4	26	179	1401
la04|nh|sum|$\frac{2}{3}$|h|max|1	700	698	**673**		3	2	2		15	94	421		4	20	158	
la04|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	628	**608**			4	3			27	93			5	19		
la04|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	710	703	**683**		3	2	2		17	92	365		4	20	152	
la04|nh|sum|$\frac{2}{3}$|nh|max|1	701	688	**667**		3	2	2		16	94	353		4	21	151	
la04|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	634	**608**			3	3			21	119			5	21		
la04|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	710	701	**680**		4	2	2		21	92	394		4	20	150	

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[a b]				# Iterations[a]				CPU[a c]				Memory[a d]			
la05|h|max|1|h|max|1	**1175**				3				26				4			
la05|h|max|1|h|max|$\frac{1}{3}$	**791**				3				28				5			
la05|h|max|1|h|sum|$\frac{1}{5}$	**1307**				3				26				4			
la05|h|max|1|nh|max|1	**1175**				3				28				5			
la05|h|max|1|nh|max|$\frac{1}{3}$	**791**				3				26				4			
la05|h|max|1|nh|sum|$\frac{1}{5}$	**1307**				3				21				4			
la05|h|max|$\frac{3}{2}$|h|max|1	**981**				3				16				4			
la05|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	**725**				3				22				5			
la05|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**1069**				3				15				4			
la05|h|max|$\frac{3}{2}$|nh|max|1	**981**				3				22				5			
la05|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	**725**				3				23				5			
la05|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	**1069**				3				14				4			
la05|h|sum|$\frac{1}{3}$|h|max|1	1209	884	-	799	2	3	1	3	15	323	648	17122	5	31	210	1704
la05|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	884	777	692	666	4	2	2	2	43	234	1648	9458	4	29	228	1616
la05|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	1286	1270	879	-	2	2	2	1	14	245	1650	5984	4	30	243	1556
la05|h|sum|$\frac{1}{3}$|nh|max|1	1155	-	835	-	2	1	3	1	14	115	2644	6086	5	29	244	1657
la05|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	777	768	692	666	6	2	2	2	75	199	1505	10038	5	28	221	1605
la05|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1238	1217	879	*838*	3	3	2	3	26	328	1619	16205	4	27	238	1708
la05|h|sum|$\frac{2}{3}$|h|max|1	**690**				2				1				5			
la05|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	**626**				2				2				5			
la05|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	**712**				2				1				5			
la05|h|sum|$\frac{2}{3}$|nh|max|1	**690**				2				2				5			
la05|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	**626**				2				1				5			
la05|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	**712**				2				2				5			
la05|nh|max|1|h|max|1	**1175**				3				27				4			
la05|nh|max|1|h|max|$\frac{1}{3}$	**791**				3				28				4			
la05|nh|max|1|h|sum|$\frac{1}{5}$	**1307**				3				27				5			
la05|nh|max|1|nh|max|1	**1175**				3				28				5			
la05|nh|max|1|nh|max|$\frac{1}{3}$	**791**				3				28				5			
la05|nh|max|1|nh|sum|$\frac{1}{5}$	**1307**				3				29				5			

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Best[a b]				# Iterations[a]				CPU[a c]				Memory[a d]			
la05|nh|max|$\frac{3}{2}$|h|max|1	**981**				3				20				4			
la05|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	**725**				3				21				4			
la05|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	**1069**				3				18				4			
la05|nh|max|$\frac{3}{2}$|nh|max|1	**981**				3				17				4			
la05|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	**725**				3				25				5			
la05|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	**1069**				2				1				4			
la05|nh|sum|$\frac{1}{3}$|h|max|1	884	-	835	-	3	1	2	1	26	105	1629	5891	4	26	197	1406
la05|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	805	692	-	666	3	2	1	2	33	224	536	10399	4	30	177	1574
la05|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	950	*889*	-	✓	3	2	1	1	27	169	597	1514	5	26	167	546
la05|nh|sum|$\frac{1}{3}$|nh|max|1	1256	*884*	835	-	4	2	2	1	45	245	1779	6960	4	29	194	1539
la05|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	792	780	745	670	4	2	2	4	51	250	2029	29849	4	29	228	2017
la05|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	1344	1306	879	-	3	2	2	1	29	268	2023	6851	4	27	215	1527
la05|nh|sum|$\frac{2}{3}$|h|max|1	**690**				2				2				5			
la05|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	**626**				2				2				4			
la05|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	**712**				2				1				4			
la05|nh|sum|$\frac{2}{3}$|nh|max|1	**690**				2				2				4			
la05|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	**626**				2				1				5			
la05|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	**712**				2				2				4			

[a] Results for different values of H (10^3, 10^4, 10^5, 10^6)
[b] Solutions proven to be optimal in bold,
solutions in italic when proven to be optimal with increased H (denoted by ✓)
[c] Sum of CPU over all iterations (s)
[d] Max Memory used over all iterations (MB)

Table A.5: *Results from iteratively finding solutions by DP for JSSPM instances (continued)*

Instance	Lower bound	# Iterations	CPU[a]	Memory[b]
la01|h|sum|$\frac{1}{3}$|h|max|1	862	10	393	163
la01|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	732	10	153	130
la01|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	934	10	403	160
la01|h|sum|$\frac{1}{3}$|nh|max|1	862	10	359	135
la01|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	732	10	160	138
la01|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	934	10	195	213
la01|nh|sum|$\frac{1}{3}$|h|max|1	901	9	582	130
la01|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	732	10	205	146
la01|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	977	9	857	146
la01|nh|sum|$\frac{1}{3}$|nh|max|1	893	10	855	139
la01|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	732	10	183	132
la01|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	954	10	867	125
la02|h|sum|$\frac{1}{3}$|h|max|1	853	9	292	136
la02|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	721	10	165	148
la02|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	909	9	349	138
la02|h|sum|$\frac{1}{3}$|nh|max|1	853	9	295	142
la02|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	721	10	142	132
la02|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	909	10	451	145
la02|nh|sum|$\frac{1}{3}$|h|max|1	882	10	979	125
la02|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	721	10	154	111
la02|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	942	10	1178	127
la02|nh|sum|$\frac{1}{3}$|nh|max|1	873	10	961	143
la02|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	721	10	167	120
la02|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	909	10	300	132
la03|h|max|1|nh|max|$\frac{1}{3}$	792	10	374	144
la03|h|sum|$\frac{1}{3}$|h|max|1	775	9	525	131
la03|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	834	10	792	131
la03|h|sum|$\frac{1}{3}$|nh|max|1	767	10	399	135
la03|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	824	10	270	163
la03|nh|sum|$\frac{1}{3}$|h|max|1	814	10	1139	133
la03|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	871	10	1207	125
la03|nh|sum|$\frac{1}{3}$|nh|max|1	805	9	980	145
la03|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	844	9	804	140
la04|h|sum|$\frac{1}{3}$|nh|max|1	810	10	729	128
la04|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	845	10	996	139
la05|h|sum|$\frac{1}{3}$|h|max|1	787	10	146	106
la05|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	659	9	154	154
la05|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	831	10	274	157
la05|h|sum|$\frac{1}{3}$|nh|max|1	787	10	277	168
la05|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	659	9	151	150
la05|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	831	9	130	135
la05|nh|sum|$\frac{1}{3}$|h|max|1	787	10	301	150
la05|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	659	9	153	132
la05|nh|sum|$\frac{1}{3}$|nh|max|1	787	10	288	161
la05|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	659	10	76	129
la05|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	831	10	262	150

[a] Sum of CPU over all iterations (s)   [b] Max Memory used over all iterations (MB)

Table A.6: *Results from iteratively finding lower bound by DP for JSSPM instances*

A

Instance	MIP Using Gurobi 5.6.3				DP		
	LB	UB	CPU (s)	UB at (s)	LB	UB	CPU (s)
ft06$\vert$h$\vert$max$\vert$1$\vert$h$\vert$max$\vert$1		**98**	24	9		**98**	47
ft06$\vert$h$\vert$max$\vert$1$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**71**	17	14		**71**	7
ft06$\vert$h$\vert$max$\vert$1$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**94**	48	14		**94**	41
ft06$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$1		**97**	76	72		**97**	11
ft06$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**71**	18	12		**71**	8
ft06$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**88**	23	19		**88**	8
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$max$\vert$1		**79**	26	23		**79**	4
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**64**	104	62		**64**	6
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**76**	100	18		**76**	5
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$max$\vert$1		**79**	32	20		**79**	2
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**64**	156	78		**64**	6
ft06$\vert$h$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**76**	27	24		**76**	5
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$max$\vert$1		**79**	26	23		**79**	4
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**64**	102	61		**64**	5
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**76**	100	18		**76**	5
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$max$\vert$1		**79**	32	20		**79**	3
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**64**	156	78		**64**	4
ft06$\vert$h$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**76**	27	23		**76**	5
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$max$\vert$1		**65**	23	18		**65**	1
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**59**	118	18		**59**	0
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**64**	15	11		**64**	1
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$max$\vert$1		**65**	22	17		**65**	0
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**59**	383	22		**59**	1
ft06$\vert$h$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**63**	35	23		**63**	0
ft06$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$max$\vert$1		**100**	22	15		**100**	54
ft06$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**71**	77	77		**71**	7
ft06$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**95**	26	26		**95**	46
ft06$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$1		**99**	28	6		**99**	52
ft06$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**71**	22	21		**71**	7
ft06$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**89**	21	18		**89**	8
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$max$\vert$1		**79**	36	35		**79**	5
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**64**	60	23		**64**	6
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**76**	47	43		**76**	5
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$max$\vert$1		**79**	25	18		**79**	5
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**64**	212	71		**64**	4
ft06$\vert$nh$\vert$max$\vert$$\frac{3}{2}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**76**	34	19		**76**	5
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$max$\vert$1		**88**	233	230		**88**	38
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**67**	150	34		**67**	6
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**86**	230	53		**86**	27
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$max$\vert$1		**86**	146	103		**86**	33
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**66**	329	114		**66**	8
ft06$\vert$nh$\vert$sum$\vert$$\frac{1}{3}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**79**	53	39		**79**	8
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$max$\vert$1		**66**	54	39		**66**	1
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$max$\vert$$\frac{1}{3}$		**60**	312	19		**60**	4
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$h$\vert$sum$\vert$$\frac{1}{5}$		**65**	48	13		**65**	1
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$max$\vert$1		**66**	41	19		**66**	2
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$max$\vert$$\frac{1}{3}$		**59**	70	53		**59**	4
ft06$\vert$nh$\vert$sum$\vert$$\frac{2}{3}$$\vert$nh$\vert$sum$\vert$$\frac{1}{5}$		**64**	42	28		**64**	1

Table A.7: *Comparison MIP and DP for JSSPM instances*

<table>
<tr><th>Instance</th><th colspan="4">MIP Using Gurobi 5.6.3</th><th colspan="3">DP</th></tr>
<tr><th></th><th>LB</th><th>UB</th><th>CPU (s)</th><th>UB at (s)</th><th>LB</th><th>UB</th><th>CPU (s)</th></tr>
<tr><td>la01|h|max|1|h|max|1</td><td>1336</td><td>1352</td><td>7200</td><td>3121</td><td></td><td>1352</td><td>230</td></tr>
<tr><td>la01|h|max|1|h|max|$\frac{1}{3}$</td><td>413</td><td>926</td><td>7200</td><td>2027</td><td></td><td>897</td><td>1887</td></tr>
<tr><td>la01|h|max|1|h|sum|$\frac{1}{5}$</td><td>1588</td><td>1604</td><td>7200</td><td>7164</td><td></td><td>1604</td><td>2016</td></tr>
<tr><td>la01|h|max|1|nh|max|1</td><td>442</td><td>1367</td><td>7200</td><td>4810</td><td></td><td>1352</td><td>266</td></tr>
<tr><td>la01|h|max|1|nh|max|$\frac{1}{3}$</td><td>413</td><td>930</td><td>7200</td><td>496</td><td></td><td>897</td><td>1976</td></tr>
<tr><td>la01|h|max|1|nh|sum|$\frac{1}{5}$</td><td>897</td><td>1604</td><td>7200</td><td>4349</td><td></td><td>1604</td><td>32</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|h|max|1</td><td>413</td><td>1254</td><td>7200</td><td>981</td><td></td><td>1254</td><td>44</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$</td><td>413</td><td>864</td><td>7200</td><td>474</td><td></td><td>864</td><td>23</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$</td><td>875</td><td>1470</td><td>7200</td><td>503</td><td></td><td>1470</td><td>25</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|nh|max|1</td><td>468</td><td>1254</td><td>7200</td><td>2555</td><td></td><td>1254</td><td>23</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$</td><td>413</td><td>883</td><td>7200</td><td>3378</td><td></td><td>864</td><td>25</td></tr>
<tr><td>la01|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$</td><td>567</td><td>1470</td><td>7200</td><td>366</td><td></td><td>1470</td><td>211</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|h|max|1</td><td>413</td><td>1001</td><td>7200</td><td>5551</td><td>862</td><td>999</td><td>18341</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$</td><td>413</td><td>780</td><td>7200</td><td>6933</td><td>732</td><td>765</td><td>14198</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$</td><td>966</td><td>1068</td><td>7200</td><td>7015</td><td>934</td><td>1107</td><td>19619</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|nh|max|1</td><td>484</td><td>993</td><td>7200</td><td>3405</td><td>862</td><td>1155</td><td>17480</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$</td><td>437</td><td>765</td><td>7200</td><td>5232</td><td>732</td><td>765</td><td>4648</td></tr>
<tr><td>la01|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$</td><td>480</td><td>1068</td><td>7200</td><td>2205</td><td>934</td><td>1285</td><td>19264</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|h|max|1</td><td>452</td><td>824</td><td>7200</td><td>2983</td><td></td><td>764</td><td>17</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$</td><td>550</td><td>699</td><td>7200</td><td>6298</td><td></td><td>699</td><td>85</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$</td><td>564</td><td>877</td><td>7200</td><td>7169</td><td></td><td>800</td><td>84</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|nh|max|1</td><td>530</td><td>800</td><td>7200</td><td>2958</td><td></td><td>764</td><td>82</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$</td><td>413</td><td>699</td><td>7200</td><td>1298</td><td></td><td>699</td><td>19</td></tr>
<tr><td>la01|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$</td><td>413</td><td>881</td><td>7200</td><td>7163</td><td></td><td>800</td><td>74</td></tr>
<tr><td>la01|nh|max|1|h|max|1</td><td>577</td><td>1369</td><td>7200</td><td>1521</td><td></td><td>1352</td><td>228</td></tr>
<tr><td>la01|nh|max|1|h|max|$\frac{1}{3}$</td><td>413</td><td>930</td><td>7200</td><td>411</td><td></td><td>897</td><td>1887</td></tr>
<tr><td>la01|nh|max|1|h|sum|$\frac{1}{5}$</td><td>515</td><td>1738</td><td>7200</td><td>994</td><td></td><td>1604</td><td>247</td></tr>
<tr><td>la01|nh|max|1|nh|max|1</td><td>413</td><td>1379</td><td>7200</td><td>3972</td><td></td><td>1352</td><td>246</td></tr>
<tr><td>la01|nh|max|1|nh|max|$\frac{1}{3}$</td><td>413</td><td>920</td><td>7200</td><td>6187</td><td></td><td>897</td><td>1897</td></tr>
<tr><td>la01|nh|max|1|nh|sum|$\frac{1}{5}$</td><td>880</td><td>1604</td><td>7200</td><td>1010</td><td></td><td>1604</td><td>249</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|h|max|1</td><td>542</td><td>1254</td><td>7200</td><td>1360</td><td></td><td>1254</td><td>235</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$</td><td>413</td><td>897</td><td>7200</td><td>6450</td><td></td><td>864</td><td>66</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$</td><td>514</td><td>1470</td><td>7200</td><td>1092</td><td></td><td>1470</td><td>45</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|nh|max|1</td><td>442</td><td>1254</td><td>7200</td><td>1872</td><td></td><td>1254</td><td>45</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$</td><td>413</td><td>864</td><td>7200</td><td>974</td><td></td><td>864</td><td>45</td></tr>
<tr><td>la01|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$</td><td>515</td><td>1470</td><td>7200</td><td>851</td><td></td><td>1470</td><td>47</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|h|max|1</td><td>413</td><td>1026</td><td>7200</td><td>2919</td><td>901</td><td>1014</td><td>18734</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$</td><td>413</td><td>765</td><td>7200</td><td>3894</td><td>732</td><td>765</td><td>3648</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$</td><td>464</td><td>1068</td><td>7200</td><td>1139</td><td>977</td><td>1068</td><td>7681</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|nh|max|1</td><td>699</td><td>993</td><td>7200</td><td>2767</td><td>893</td><td>999</td><td>18982</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$</td><td>521</td><td>775</td><td>7200</td><td>1565</td><td>732</td><td>765</td><td>17041</td></tr>
<tr><td>la01|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$</td><td>429</td><td>1068</td><td>7200</td><td>3266</td><td>954</td><td>1107</td><td>21812</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|h|max|1</td><td>575</td><td>805</td><td>7200</td><td>1428</td><td></td><td>764</td><td>11</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$</td><td>413</td><td>723</td><td>7200</td><td>3708</td><td></td><td>699</td><td>69</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$</td><td>540</td><td>800</td><td>7200</td><td>687</td><td></td><td>800</td><td>11</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|nh|max|1</td><td>542</td><td>764</td><td>7200</td><td>4924</td><td></td><td>764</td><td>18</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$</td><td>413</td><td>743</td><td>7200</td><td>7084</td><td></td><td>699</td><td>78</td></tr>
<tr><td>la01|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$</td><td></td><td>800</td><td>4474</td><td>2801</td><td></td><td>800</td><td>9</td></tr>
</table>

Table A.7: *Comparison MIP and DP for JSSPM instances (continued)*

A

Instance	MIP Using Gurobi 5.6.3				DP		
	LB	UB	CPU (s)	UB at (s)	LB	UB	CPU (s)
la02|h|max|1|h|max|1	394	1308	7200	5803		**1249**	197
la02|h|max|1|h|max|$\frac{1}{3}$	394	889	7200	6107		**853**	211
la02|h|max|1|h|sum|$\frac{1}{5}$	487	1587	7200	1185		**1417**	286
la02|h|max|1|nh|max|1	560	1381	7200	2121		**1249**	233
la02|h|max|1|nh|max|$\frac{1}{3}$	394	929	7200	7175		**853**	241
la02|h|max|1|nh|sum|$\frac{1}{5}$	394	1557	7200	5391		**1417**	53
la02|h|max|$\frac{3}{2}$|h|max|1	394	1150	7200	2784		**1051**	18
la02|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	394	853	7200	5553		**787**	33
la02|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	407	1397	7200	2357		**1163**	123
la02|h|max|$\frac{3}{2}$|nh|max|1	889	1075	7200	6635		**1051**	163
la02|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	394	818	7200	6711		**787**	22
la02|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	394	1245	7200	6651		**1163**	152
la02|h|sum|$\frac{1}{3}$|h|max|1	394	993	7200	7187	853	972	20587
la02|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	394	761	7200	6926	721	754	4407
la02|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	933	1036	7200	2611	909	1056	22223
la02|h|sum|$\frac{1}{3}$|nh|max|1	394	1051	7200	5009	853	972	19508
la02|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	394	822	7200	1985	721	754	5943
la02|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	818	1036	7200	3584	909	1307	23160
la02|h|sum|$\frac{2}{3}$|h|max|1	588	754	7200	2006		**754**	12
la02|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	394	754	7200	4350		**688**	11
la02|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	627	861	7200	5105		**782**	42
la02|h|sum|$\frac{2}{3}$|nh|max|1	623	754	7200	4865		**754**	11
la02|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	394	721	7200	3967		**688**	12
la02|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	645	782	7200	6903		**782**	125
la02|nh|max|1|h|max|1	407	1465	7200	7142		**1260**	1341
la02|nh|max|1|h|max|$\frac{1}{3}$	394	916	7200	5098		**853**	269
la02|nh|max|1|h|sum|$\frac{1}{5}$	435	1613	7200	734		**1428**	1820
la02|nh|max|1|nh|max|1	394	1407	7200	6801		**1249**	242
la02|nh|max|1|nh|max|$\frac{1}{3}$	394	886	7200	3892		**853**	465
la02|nh|max|1|nh|sum|$\frac{1}{5}$	394	1544	7200	445		**1417**	161
la02|nh|max|$\frac{3}{2}$|h|max|1	414	1198	7200	7136		**1051**	144
la02|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	394	837	7200	5507		**787**	1236
la02|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	394	1329	7200	7175		**1163**	1239
la02|nh|max|$\frac{3}{2}$|nh|max|1	394	1180	7200	3264		**1051**	147
la02|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	394	820	7200	7015		**787**	1140
la02|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	471	1163	7200	2880		**1163**	21
la02|nh|sum|$\frac{1}{3}$|h|max|1	494	1063	7200	6291	882	988	23481
la02|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	394	765	7200	4697	721	754	5772
la02|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	491	1112	7200	4527	942	1056	9993
la02|nh|sum|$\frac{1}{3}$|nh|max|1	394	1080	7200	1655	873	988	22309
la02|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	394	824	7200	1120	721	754	4998
la02|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	761	1036	7200	1830	909	1036	8670
la02|nh|sum|$\frac{2}{3}$|h|max|1	599	769	7200	5918		**754**	90
la02|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	394	753	7200	6334		**688**	19
la02|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	394	961	7200	6998		**782**	106
la02|nh|sum|$\frac{2}{3}$|nh|max|1	394	794	7200	6751		**754**	92
la02|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	394	754	7200	2158		**688**	19
la02|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	394	1036	7200	3941		**782**	36

Table A.7: *Comparison MIP and DP for JSSPM instances (continued)*

A

Instance	MIP Using Gurobi 5.6.3				DP		
	LB	UB	CPU (s)	UB at (s)	LB	UB	CPU (s)
la03|h|max|1|h|max|1	497	1316	7200	6839		**1158**	26
la03|h|max|1|h|max|$\frac{1}{3}$	349	805	7200	5496		**800**	5904
la03|h|max|1|h|sum|$\frac{1}{5}$	358	1506	7200	4784		**1320**	72
la03|h|max|1|nh|max|1	349	1211	7200	5053		**1146**	42
la03|h|max|1|nh|max|$\frac{1}{3}$	349	834	7200	5944	792	798	6730
la03|h|max|1|nh|sum|$\frac{1}{5}$	439	1464	7200	7035		**1320**	270
la03|h|max|$\frac{3}{2}$|h|max|1	349	1043	7200	3115		**952**	155
la03|h|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	349	752	7200	7198		**721**	7668
la03|h|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	526	1178	7200	1671		**1060**	19
la03|h|max|$\frac{3}{2}$|nh|max|1	368	1033	7200	3649		**944**	31
la03|h|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	349	783	7200	5465		**717**	8077
la03|h|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	384	1082	7200	4617		**1060**	176
la03|h|sum|$\frac{1}{3}$|h|max|1	349	861	7200	3670	775	882	18663
la03|h|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	349	748	7200	1648		**681**	9498
la03|h|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	494	946	7200	5994	834	963	20488
la03|h|sum|$\frac{1}{3}$|nh|max|1	433	855	7200	4699	767	876	19309
la03|h|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	530	708	7200	6862		**678**	9119
la03|h|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	349	1060	7200	2238	824	963	20008
la03|h|sum|$\frac{2}{3}$|h|max|1	356	770	7200	6810		**688**	1198
la03|h|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	349	717	7200	4936		**628**	71
la03|h|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	349	767	7200	2875		**715**	1705
la03|h|sum|$\frac{2}{3}$|nh|max|1	349	732	7200	4514		**686**	1259
la03|h|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	349	708	7200	6596		**627**	71
la03|h|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	349	735	7200	6764		**706**	96
la03|nh|max|1|h|max|1	388	1369	7200	6380		**1249**	50
la03|nh|max|1|h|max|$\frac{1}{3}$	349	885	7200	6549		**829**	250
la03|nh|max|1|h|sum|$\frac{1}{5}$	349	1438	7200	5138		**1438**	177
la03|nh|max|1|nh|max|1	489	1300	7200	4793		**1235**	29
la03|nh|max|1|nh|max|$\frac{1}{3}$	401	842	7200	4659		**822**	255
la03|nh|max|1|nh|sum|$\frac{1}{5}$	390	1438	7200	3876		**1438**	292
la03|nh|max|$\frac{3}{2}$|h|max|1	349	1043	7200	4165		**953**	38
la03|nh|max|$\frac{3}{2}$|h|max|$\frac{1}{3}$	349	789	7200	1661		**723**	1343
la03|nh|max|$\frac{3}{2}$|h|sum|$\frac{1}{5}$	1032	1151	7200	6556		**1061**	37
la03|nh|max|$\frac{3}{2}$|nh|max|1	349	1054	7200	3052		**945**	24
la03|nh|max|$\frac{3}{2}$|nh|max|$\frac{1}{3}$	349	786	7200	444		**722**	1772
la03|nh|max|$\frac{3}{2}$|nh|sum|$\frac{1}{5}$	350	1061	7200	3691		**1061**	212
la03|nh|sum|$\frac{1}{3}$|h|max|1	783	870	7200	3411	814	882	17800
la03|nh|sum|$\frac{1}{3}$|h|max|$\frac{1}{3}$	349	712	7200	7084		**681**	8977
la03|nh|sum|$\frac{1}{3}$|h|sum|$\frac{1}{5}$	586	1024	7200	5808	871	974	20413
la03|nh|sum|$\frac{1}{3}$|nh|max|1	435	910	7200	3158	805	887	19767
la03|nh|sum|$\frac{1}{3}$|nh|max|$\frac{1}{3}$	349	759	7200	3721		**678**	8791
la03|nh|sum|$\frac{1}{3}$|nh|sum|$\frac{1}{5}$	376	943	7200	4602	844	963	20862
la03|nh|sum|$\frac{2}{3}$|h|max|1	358	758	7200	5874		**688**	1231
la03|nh|sum|$\frac{2}{3}$|h|max|$\frac{1}{3}$	349	712	7200	4525		**628**	79
la03|nh|sum|$\frac{2}{3}$|h|sum|$\frac{1}{5}$	386	824	7200	1299		**715**	1553
la03|nh|sum|$\frac{2}{3}$|nh|max|1	460	694	7200	5270		**686**	1383
la03|nh|sum|$\frac{2}{3}$|nh|max|$\frac{1}{3}$	431	690	7200	6902		**627**	87
la03|nh|sum|$\frac{2}{3}$|nh|sum|$\frac{1}{5}$	349	836	7200	7192		**706**	91

Table A.7: *Comparison MIP and DP for JSSPM instances (continued)*

A

Instance	MIP Using Gurobi 5.6.3				DP		
	LB	UB	CPU (s)	UB at (s)	LB	UB	CPU (s)
la04$\vert$h$\vert$max$\vert$1$\vert$h$\vert$max$\vert$1	412	1211	7200	1078		**1126**	186
la04$\vert$h$\vert$max$\vert$1$\vert$h$\vert$max$\vert\frac{1}{3}$	369	895	7200	4910		**753**	388
la04$\vert$h$\vert$max$\vert$1$\vert$h$\vert$sum$\vert\frac{1}{5}$	446	1286	7200	4720		**1186**	186
la04$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$1	534	1162	7200	4966		**1078**	541
la04$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$max$\vert\frac{1}{3}$	369	842	7200	467		**740**	443
la04$\vert$h$\vert$max$\vert$1$\vert$nh$\vert$sum$\vert\frac{1}{5}$	369	1277	7200	7170		**1186**	190
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$h$\vert$max$\vert$1	369	1026	7200	6873		**959**	906
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$h$\vert$max$\vert\frac{1}{3}$	369	732	7200	3309		**712**	5059
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$h$\vert$sum$\vert\frac{1}{5}$	369	1150	7200	6664		**999**	1261
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$max$\vert$1	369	1019	7200	6860		**959**	149
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$max$\vert\frac{1}{3}$	369	720	7200	6307		**707**	4456
la04$\vert$h$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$sum$\vert\frac{1}{5}$	399	999	7200	5689		**999**	153
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$max$\vert$1	575	873	7200	6573		**861**	8988
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$max$\vert\frac{1}{3}$	369	743	7200	6793		**666**	1418
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$sum$\vert\frac{1}{5}$	516	968	7200	2259		**891**	9419
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$max$\vert$1	369	933	7200	1431	810	861	6229
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$max$\vert\frac{1}{3}$	538	709	7200	6520		**666**	1039
la04$\vert$h$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$sum$\vert\frac{1}{5}$	380	946	7200	5932	845	891	13296
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$max$\vert$1	369	763	7200	7178		**673**	478
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$max$\vert\frac{1}{3}$	492	626	7200	4319		**608**	135
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$sum$\vert\frac{1}{5}$	488	737	7200	6762		**683**	501
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$max$\vert$1	369	722	7200	6331		**667**	436
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$max$\vert\frac{1}{3}$	412	648	7200	5708		**608**	135
la04$\vert$h$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$sum$\vert\frac{1}{5}$	369	783	7200	6391		**680**	567
la04$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$max$\vert$1	369	1167	7200	6923		**1126**	185
la04$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$max$\vert\frac{1}{3}$	369	798	7200	6220		**756**	1253
la04$\vert$nh$\vert$max$\vert$1$\vert$h$\vert$sum$\vert\frac{1}{5}$	477	1394	7200	1068		**1186**	211
la04$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$max$\vert$1	373	1373	7200	1099		**1080**	1422
la04$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$max$\vert\frac{1}{3}$	369	831	7200	3328		**753**	2005
la04$\vert$nh$\vert$max$\vert$1$\vert$nh$\vert$sum$\vert\frac{1}{5}$	369	1243	7200	6984		**1186**	44
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$h$\vert$max$\vert$1	379	1014	7200	6848		**972**	2131
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$h$\vert$max$\vert\frac{1}{3}$	369	743	7200	5377		**727**	6870
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$h$\vert$sum$\vert\frac{1}{5}$	369	1158	7200	4648		**1012**	2130
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$max$\vert$1	369	1057	7200	2798		**959**	873
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$max$\vert\frac{1}{3}$	369	753	7200	6922		**716**	6621
la04$\vert$nh$\vert$max$\vert\frac{3}{2}\vert$nh$\vert$sum$\vert\frac{1}{5}$	664	1165	7200	6958		**999**	740
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$max$\vert$1	369	1026	7200	6839		**861**	9855
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$max$\vert\frac{1}{3}$	369	822	7200	2371		**671**	1894
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$h$\vert$sum$\vert\frac{1}{5}$	640	922	7200	6569		**891**	8063
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$max$\vert$1	369	917	7200	2455		**861**	9722
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$max$\vert\frac{1}{3}$	369	699	7200	6981		**666**	2431
la04$\vert$nh$\vert$sum$\vert\frac{1}{3}\vert$nh$\vert$sum$\vert\frac{1}{5}$	369	945	7200	4575		**891**	9354
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$max$\vert$1	369	791	7200	4700		**673**	530
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$max$\vert\frac{1}{3}$	369	649	7200	5322		**608**	120
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$h$\vert$sum$\vert\frac{1}{5}$	519	695	7200	7175		**683**	474
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$max$\vert$1	369	771	7200	6708		**667**	463
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$max$\vert\frac{1}{3}$	369	708	7200	1624		**608**	140
la04$\vert$nh$\vert$sum$\vert\frac{2}{3}\vert$nh$\vert$sum$\vert\frac{1}{5}$	431	746	7200	4587		**680**	507

Table A.7: *Comparison MIP and DP for JSSPM instances (continued)*

A

Instance	MIP Using Gurobi 5.6.3				DP		
	LB	UB	CPU (s)	UB at (s)	LB	UB	CPU (s)
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{max}\mid 1$	1144	1175	7200	5474		**1175**	26
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	791	7200	897		**791**	28
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	380	1307	7200	5513		**1307**	26
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	766	1175	7200	663		**1175**	28
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	791	7200	4212		**791**	26
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$		**1307**	3096	855		**1307**	21
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	380	981	7200	6103		**981**	16
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	725	7200	1941		**725**	22
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	536	1069	7200	454		**1069**	15
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	402	981	7200	778		**981**	22
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	725	7200	563		**725**	23
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	526	1069	7200	1259		**1069**	14
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	380	884	7200	3479	787	799	18254
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	692	7200	1581	659	666	11537
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	380	950	7200	307	831	879	8167
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	391	849	7200	3999	787	835	9136
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	692	7200	4804	659	666	11968
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$		**838**	7088	7076	831	838	18308
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	436	690	7200	3346		**690**	1
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	626	7200	5677		**626**	2
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$		**712**	6808	2127		**712**	1
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	380	706	7200	5997		**690**	2
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	725	7200	4025		**626**	1
$\mathrm{la05}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	380	928	7200	597		**712**	2
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{max}\mid 1$	380	1262	7200	3404		**1175**	27
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	791	7200	6479		**791**	28
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	1269	1311	7200	6591		**1307**	27
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	394	1175	7200	4074		**1175**	28
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	791	7200	3661		**791**	28
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid 1\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	473	1307	7200	929		**1307**	29
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	405	1017	7200	4576		**981**	20
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	730	7200	1401		**725**	21
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	842	1069	7200	1028		**1069**	18
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	380	981	7200	2238		**981**	17
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	725	7200	1583		**725**	25
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{3}{2}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	790	1069	7200	2030		**1069**	1
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	507	884	7200	2416	787	835	7952
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	692	7200	2706	659	666	11345
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	380	950	7200	1596		**889**	2307
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	495	883	7200	7159	787	835	9317
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	692	7200	935	659	670	32255
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{3}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	380	950	7200	1371	831	879	9433
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{max}\mid 1$	405	690	7200	6406		**690**	2
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{max}\mid \frac{1}{3}$	380	626	7200	6549		**626**	2
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{h}\mid\mathrm{sum}\mid \frac{1}{5}$	490	712	7200	5034		**712**	1
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid 1$	652	690	7200	1760		**690**	2
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{max}\mid \frac{1}{3}$	380	659	7200	748		**626**	1
$\mathrm{la05}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{2}{3}\mid\mathrm{nh}\mid\mathrm{sum}\mid \frac{1}{5}$	380	1069	7200	512		**712**	2

Table A.7: *Comparison MIP and DP for JSSPM instances (continued)*

Appendix B

Job Shop Scheduling Problem Instances

This appendix provides the set of JSSP benchmark instances used in this dissertation. It also gives the values of the upper and lower bounds known at the moment of our experiments. For these instances we did our best to find the origin of the best known upper bound and lower bound.

The instances can be obtained from [13,112,108]. Information about the current upper and lower bounds are mostly obtained from [70,112,108]. Note, the newly found lower bounds given in table 5.5 are not included in the tables below. Also the brand new results of Vilím, Laborie, and Shaw [120], see also [119], are not incorporated in these tables as these bounds were not used in the generation of the computational results of this dissertation.

All the bounds in this appendix can be found on my web site with JSSP instances. Including the new results mentioned above. This web site can be found at `http://jobshop.jjvh.nl` [67]. All optimal solutions that are found in the course of this dissertation are also included on this web site.

B

Fisher and Thompson

H. Fisher and G. L. Thompson. *Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules.* In: *Industrial Scheduling*, 15: *225–251*. ed. by J. F. Muth and G. L. Thompson. Prentice Hall, 1963

Instance	# jobs	# machines	Lower bound	Upper bound
ft06	6	6	55[46] [a]	**55**[46] [a]
ft10	10	10	930[24] [b]	**930**[c]
ft20	20	5	1165[86]	**1165**[86]

[24] Carlier and Pinson (1989)
[46] Florian, Trepant, and McMahon (1971)
[86] McMahon and Florian (1975)
[a] Using algorithms of Schrage [106] and Balas [10]
[b] Achieved in 1986 [see 2]
[c] B.J. Lageweg (1984) [see 75]

Table B.1: *Instances of Fisher and Thompson [44]*

Lawrence

S. Lawrence. *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement).* Carnegie-Mellon University, 1984

Instance	# jobs	# machines	Lower bound	Upper bound
la01	10	5	666[2]	**666**[2]
la02	10	5	655[2]	**655**[85]
la03	10	5	597[7]	**597**[85]
la04	10	5	590[7]	**590**[85]
la05	10	5	593[2]	**593**[2]
la06	15	5	926[2]	**926**[2]
la07	15	5	890[2]	**890**[2]
la08	15	5	863[2]	**863**[2]
la09	15	5	951[2]	**951**[2]
la10	15	5	958[2]	**958**[85]

[2] Adams, Balas, and Zawack (1988)
[7] Applegate and Cook (1991)
[85] Matsuo, Suh, and Sullivan (1988)

Table B.2: *Instances of Lawrence [79]*

B

Instance	# jobs	# machines	Lower bound	Upper bound
la11	20	5	1222[2]	**1222**[2]
la12	20	5	1039[2]	**1039**[2]
la13	20	5	1150[2]	**1150**[2]
la14	20	5	1292[2]	**1292**[2]
la15	20	5	1207[2]	**1207**[2]
la16	10	10	945[25]	**945**[25]
la17	10	10	784[25]	**784**[85]
la18	10	10	848[7]	**848**[85]
la19	10	10	842[7]	**842**[85]
la20	10	10	902[7]	**902**[7]
la21	15	10	1046[118]	**1046**[118]
la22	15	10	927[7]	**927**[85]
la23	15	10	1032[2]	**1032**[2]
la24	15	10	935[7]	**935**[7]
la25	15	10	977[7]	**977**[7]
la26	20	10	1218[2]	**1218**[85]
la27	20	10	1235[2]	**1235**[26]
la28	20	10	1216[2]	**1216**[7]
la29	20	10	1152[84]	**1152**[84]
la30	20	10	1355[2]	**1355**[2]
la31	30	10	1784[2]	**1784**[2]
la32	30	10	1850[2]	**1850**[2]
la33	30	10	1719[2]	**1719**[2]
la34	30	10	1721[2]	**1721**[2]
la35	30	10	1888[2]	**1888**[2]
la36	15	15	1268[25]	**1268**[25]
la37	15	15	1397[7]	**1397**[7]
la38	15	15	1196[118]	**1196**[90]
la39	15	15	1233[7]	**1233**[7]
la40	15	15	1222[7]	**1222**[7]

[2] Adams, Balas, and Zawack (1988)
[7] Applegate and Cook (1991)
[25] Carlier and Pinson (1990)
[26] Carlier and Pinson (1994)
[84] Martin (1996)
[85] Matsuo, Suh, and Sullivan (1988)
[90] Nowicki and Smutnicki (1996)
[118] Vaessens, Aarts, and Lenstra (1996)

Table B.2: *Instances of Lawrence [79] (continued)*

Adams, Balas, and Zawack

Joseph Adams, Egon Balas, and Daniel Zawack. *The Shifting Bottleneck Procedure for Job Shop Scheduling.* Management Science, 34.3: *391–401*, 1988

Instance	# jobs	# machines	Lower bound	Upper bound
abz5	10	10	1234[7]	**1234**[7]
abz6	10	10	943[7]	**943**[2]
abz7	20	15	656[84]	**656**[84]
abz8	20	15	646[21]	665[84]
abz9	20	15	678[73]	**678**[125]

[2] Adams, Balas, and Zawack (1988)
[7] Applegate and Cook (1991)
[21] Brinkkötter and Brucker (2001)
[73] Koshimura, Nabeshima, Fujita, and Hasegawa (2010)
[84] Martin (1996)
[125] Zhang, Li, Rao, and Guan (2008)

Table B.3: *Instances of Adams, Balas, and Zawack [2]*

Applegate and Cook

David Applegate and William Cook. *A Computational Study of the Job-Shop Scheduling Problem.* ORSA Journal on Computing, 3.2: *149–156*, 1991

Instance	# jobs	# machines	Lower bound	Upper bound
orb01	10	10	1059[7]	**1059**[7]
orb02	10	10	888[7]	**888**[7]
orb03	10	10	1005[7]	**1005**[7]
orb04	10	10	1005[7]	**1005**[7]
orb05	10	10	887[7]	**887**[7]
orb06	10	10	1010[a]	**1010**[a]
orb07	10	10	397[a]	**397**[a]
orb08	10	10	899[a]	**899**[a]
orb09	10	10	934[a]	**934**[a]
orb10	10	10	944[a]	**944**[a]

[7] Applegate and Cook (1991)
[a] R.J.M. Vaessens using algorithms of [7] (1994) [see 70]

Table B.4: *Instances of Applegate and Cook [7]*

B

Storer, Wu, and Vaccari

Robert H. Storer, S. David Wu, and Renzo Vaccari. *New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling*. Management Science, 38.10: *1495–1509*, 1992

Instance	# jobs	# machines	Lower bound	Upper bound
swv01	20	10	1407[84]	**1407**[84]
swv02	20	10	1475[84]	**1475**[84]
swv03	20	10	1398[21]	**1398**[117]
swv04	20	10	1450[117]	1467[a]
swv05	20	10	1424[84]	**1424**[84]
swv06	20	15	1591[117]	1671[95]
swv07	20	15	1447[21]	1594[89]
swv08	20	15	1641[21]	1752[29]
swv09	20	15	1605[21]	1655[89]
swv10	20	15	1632[21]	1743[56]
swv11	50	10	2983[117]	**2983**[92]
swv12	50	10	2972[117]	2977[95]
swv13	50	10	3104[117]	**3104**[114]
swv14	50	10	2968[11]	**2968**[11]
swv15	50	10	2885[117]	**2885**[95]
swv16	50	10	2924[110]	**2924**[110]
swv17	50	10	2794[110]	**2794**[110]
swv18	50	10	2852[110]	**2852**[110]
swv19	50	10	2843[110]	**2843**[110]
swv20	50	10	2823[110]	**2823**[110]

[11] Balas and Vazacopoulos (1994)
[21] Brinkkötter and Brucker (2001)
[29] Cheng, Peng, and Lü (2013)
[56] Gonçalves and Resende (2014)
[84] Martin (1996)
[89] Nagata and Tojo (2009)
[92] Nowicki and Smutnicki (2005)
[95] Peng, Lü, and Cheng (2015)
[110] Storer, Wu, and Vaccari (1992)
[114] Thomsen (1997)
[117] Vaessens (1996)
[a] O. V. Shylo (2013) [see 108]

Table B.5: *Instances of Storer, Wu, and Vaccari [110]*

B

Yamada and Nakano

Takeshi Yamada and Ryohei Nakano. *A genetic algorithm applicable to large-scale job-shop instances.* In: *Parallel instance solving from nature 2*: *281–290.* ed. by Reinhard Männer and Bernard Manderick. Elsevier, 1992.

Instance	# jobs	# machines	Lower bound	Upper bound
yn1	20	20	884[73]	**884**[125]
yn2	20	20	870[21]	904[56]
yn3	20	20	840[21]	892[92]
yn4	20	20	920[21]	968[114]

[21] Brinkkötter and Brucker (2001)
[56] Gonçalves and Resende (2014)
[73] Koshimura, Nabeshima, Fujita, and Hasegawa (2010)
[92] Nowicki and Smutnicki (2005)
[114] Thomsen (1997)
[125] Zhang, Li, Rao, and Guan (2008)

Table B.6: *Instances of Yamada and Nakano [123]*

Taillard

E.D. Taillard. *Benchmarks for basic scheduling problems.* European Journal of Operational Research, 64.2: *278–285*, 1993

Instance	# jobs	# machines	Lower bound	Upper bound
ta01	15	15	1231[113,111]	**1231**[113,111]
ta02	15	15	1244[a]	**1244**[90]
ta03	15	15	1218[21]	**1218**[11]
ta04	15	15	1175[21]	**1175**[b]
ta05	15	15	1224[21]	**1224**[21]
ta06	15	15	1238[21]	**1238**[21]
ta07	15	15	1227[21]	**1227**[21]
ta08	15	15	1217[21]	**1217**[11]
ta09	15	15	1274[21]	**1274**[11]
ta10	15	15	1241[a]	**1241**[11]

[11] Balas and Vazacopoulos (1994)
[21] Brinkkötter and Brucker (2001)
[90] Nowicki and Smutnicki (1996)
[111] Taillard (1993)
[113] Taillard (1994)
[a] R.J.M. Vaessens (1995) [see 112]
[b] M. Wennink (1995) [see 112]

Table B.7: *Instances of Taillard [111]*

B

Instance	# jobs	# machines	Lower bound	Upper bound
ta11	20	15	1323[a]	1357[93]
ta12	20	15	1351[a]	1367[11]
ta13	20	15	1282[a]	1342[66]
ta14	20	15	1345[b]	**1345**[90]
ta15	20	15	1304[a]	1339[93]
ta16	20	15	1304[c]	1360[66] [d]
ta17	20	15	1462[a]	**1462**[92] [e]
ta18	20	15	1369[b]	1396[11]
ta19	20	15	1304[c]	1332[93]
ta20	20	15	1318[a]	1348[93]
ta21	20	20	1573[c]	1642[14]
ta22	20	20	1542[c]	1600[92] [f]
ta23	20	20	1474[c]	1557[92] [f]
ta24	20	20	1606[c]	1644[14]
ta25	20	20	1518[c]	1595[92] [e]
ta26	20	20	1558[c]	1643[14]
ta27	20	20	1617[c]	1680[92] [f]
ta28	20	20	1591[b]	1603[125]
ta29	20	20	1525[c]	1625[g]
ta30	20	20	1485[c]	1584[92] [f]
ta31	30	15	1764[111]	**1764**[h]
ta32	30	15	1774[111]	1784[i]
ta33	30	15	1778[b]	1791[94]
ta34	30	15	1828[111]	1829[92] [f]
ta35	30	15	2007[b]	**2007**[113,111]
ta36	30	15	1819[b]	**1819**[h]
ta37	30	15	1771[111]	**1771**[95]
ta38	30	15	1673[111]	**1673**[d]
ta39	30	15	1795[b]	**1795**[h]
ta40	30	15	1631[b]	1669[56]

[11] Balas and Vazacopoulos (1994)
[14] Beck, Feng, and Watson (2011)
[56] Gonçalves and Resende (2014)
[66] Henning (2002)
[90] Nowicki and Smutnicki (1996)
[92] Nowicki and Smutnicki (2005)
[93] Pardalos and Shylo (2006)
[94] Pardalos, Shylo, and Vazacopoulos (2010)
[95] Peng, Lü, and Cheng (2015)
[111] Taillard (1993)
[113] Taillard (1994)
[125] Zhang, Li, Rao, and Guan (2008)

[a] R. Schilham (2000) [see 112]
[b] R.J.M. Vaessens (1995) [see 112]
[c] Gharbi and Labidi (2011) using algorithms described in [51] [see 112]
[d] A. Henning (2000) [see 112]
[e] Achieved in 2002 [see 112]
[f] Achieved in 2001 [see 112]
[g] E. Aarts (1996) [see 112]
[h] E. Aarts, H. ten Eikelder, J.K. Lenstra and R. Schilham (1999) [see 112]
[i] In [94] (2010) [see 108]. However 1790 is mentioned. 1785 is found in [56]

Table B.7: *Instances of Taillard [111] (continued)*

Instance	# jobs	# machines	Lower bound	Upper bound
ta41	30	20	1876[a]	2005[88]
ta42	30	20	1867[b]	1937[56]
ta43	30	20	1809[b]	1846[95]
ta44	30	20	1927[b]	1979[88]
ta45	30	20	1997[b]	2000[92] [c]
ta46	30	20	1940[111]	2004[56]
ta47	30	20	1789[b]	1889[95]
ta48	30	20	1912[b]	1941[d]
ta49	30	20	1915[b]	1961[88]
ta50	30	20	1807[b]	1923[d]
ta51	50	15	2760[113,111]	**2760**[113,111]
ta52	50	15	2756[113,111]	**2756**[113,111]
ta53	50	15	2717[113,111]	**2717**[113,111]
ta54	50	15	2839[113,111]	**2839**[113,111]
ta55	50	15	2679[111]	**2679**[90]
ta56	50	15	2781[113,111]	**2781**[113,111]
ta57	50	15	2943[113,111]	**2943**[113,111]
ta58	50	15	2885[113,111]	**2885**[113,111]
ta59	50	15	2655[113,111]	**2655**[113,111]
ta60	50	15	2723[113,111]	**2723**[113,111]
ta61	50	20	2868[111]	**2868**[90]
ta62	50	20	2869[b]	**2869**[e]
ta63	50	20	2755[111]	**2755**[90]
ta64	50	20	2702[11]	**2702**[90]
ta65	50	20	2725[111]	**2725**[90]
ta66	50	20	2845[111]	**2845**[90]
ta67	50	20	2825[b]	**2825**[69]
ta68	50	20	2784[11]	**2784**[90]
ta69	50	20	3071[111]	**3071**[90]
ta70	50	20	2995[111]	**2995**[90]

[11] Balas and Vazacopoulos (1994)
[69] Jain (1998)
[56] Gonçalves and Resende (2014)
[88] Nagata and Ono (2013)
[90] Nowicki and Smutnicki (1996)
[92] Nowicki and Smutnicki (2005)
[95] Peng, Lü, and Cheng (2015)
[111] Taillard (1993)
[113] Taillard (1994)

[a] Gharbi and Labidi (2011) using algorithms described in [51] [see 112]
[b] R.J.M. Vaessens (1995) [see 112]
[c] Achieved in 2001 [see 112]
[d] O. V. Shylo (2013) [see 108]
[e] J. P. Caldeira (2003) [see 112]

Table B.7: *Instances of Taillard [111] (continued)*

B

Instance	# jobs	# machines	Lower bound	Upper bound
ta71	100	20	5464[113,111]	**5464**[113,111]
ta72	100	20	5181[113,111]	**5181**[113,111]
ta73	100	20	5568[113,111]	**5568**[113,111]
ta74	100	20	5339[113,111]	**5339**[113,111]
ta75	100	20	5392[113,111]	**5392**[113,111]
ta76	100	20	5342[113,111]	**5342**[113,111]
ta77	100	20	5436[113,111]	**5436**[113,111]
ta78	100	20	5394[113,111]	**5394**[113,111]
ta79	100	20	5358[113,111]	**5358**[113,111]
ta80	100	20	5183[111]	**5183**[90]

[90] Nowicki and Smutnicki (1996)
[111] Taillard (1993)
[113] Taillard (1994)

Table B.7: *Instances of Taillard [111] (continued)*

Demirkol, Mehta, and Uzsoy

Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. *Benchmarks for shop scheduling problems*. European Journal of Operational Research, 109.1: *137–141*, 1998

Instance	# jobs	# machines	Lower bound	Upper bound
dmu01	20	15	2501[21]	2563[66]
dmu02	20	15	2651[21]	2706[66]
dmu03	20	15	2731[21]	**2731**[21]
dmu04	20	15	2601[21]	2669[21]
dmu05	20	15	2749[21]	**2749**[21]
dmu06	20	20	2998[a]	3244[94]
dmu07	20	20	2815[a]	3046[94]
dmu08	20	20	3051[a]	3188[94]
dmu09	20	20	2956[a]	3092[66]
dmu10	20	20	2858[a]	2984[93]

[21] Brinkkötter and Brucker (2001)
[66] Henning (2002)
[93] Pardalos and Shylo (2006)
[94] Pardalos, Shylo, and Vazacopoulos (2010)
[a] Gharbi and Labidi (2011) using algorithms described in [51] [see 108]

Table B.8: *Instances of Demirkol, Mehta, and Uzsoy [38]*

B

Instance	# jobs	# machines	Lower bound	Upper bound
dmu11	30	15	3395[36,37,38]	3430[95]
dmu12	30	15	3481[36,37,38]	3495[95]
dmu13	30	15	3681[36,37,38]	**3681**[124]
dmu14	30	15	3394[36,37,38]	**3394**[91]
dmu15	30	15	3343[a]	**3343**[69]
dmu16	30	20	3734[a]	3751[56]
dmu17	30	20	3709[a]	3814[b]
dmu18	30	20	3844[36,37,38]	**3844**[56]
dmu19	30	20	3669[a]	3768[95]
dmu20	30	20	3604[36,37,38]	3710[95]
dmu21	40	15	4380[36,37,38]	**4380**[69]
dmu22	40	15	4725[36,37,38]	**4725**[69]
dmu23	40	15	4668[36,37,38]	**4668**[69]
dmu24	40	15	4648[36,37,38]	**4648**[69]
dmu25	40	15	4164[36,37,38]	**4164**[69]
dmu26	40	20	4647[36,37,38]	**4647**[124]
dmu27	40	20	4848[36,37,38]	**4848**[91]
dmu28	40	20	4692[36,37,38]	**4692**[69]
dmu29	40	20	4691[36,37,38]	**4691**[91]
dmu30	40	20	4732[36,37,38]	**4732**[91]
dmu31	50	15	5640[36,37,38]	**5640**[69]
dmu32	50	15	5927[36,37,38]	**5927**[36,37,38]
dmu33	50	15	5728[36,37,38]	**5728**[36,37,38]
dmu34	50	15	5385[36,37,38]	**5385**[36,37,38]
dmu35	50	15	5635[36,37,38]	**5635**[36,37,38]
dmu36	50	20	5621[36,37,38]	**5621**[69]
dmu37	50	20	5851[36,37,38]	**5851**[91]
dmu38	50	20	5713[36,37,38]	**5713**[69]
dmu39	50	20	5747[36,37,38]	**5747**[69]
dmu40	50	20	5577[36,37,38]	**5577**[69]

[36] Demirkol, Mehta, and Uzsoy (1996)
[37] Demirkol, Mehta, and Uzsoy (1997)
[38] Demirkol, Mehta, and Uzsoy (1998)
[56] Gonçalves and Resende (2014)
[69] Jain (1998)
[91] Nowicki and Smutnicki (2001)
[95] Peng, Lü, and Cheng (2015)
[124] Zhang, Li, Guan, and Rao (2007)
[a] Gharbi and Labidi (2011) using algorithms described in [51] [see 108]
[b] O. V. Shylo (2013) [see 108]

Table B.8: *Instances of Demirkol, Mehta, and Uzsoy [38] (continued)*

B

Instance	# jobs	# machines	Lower bound	Upper bound
dmu41	20	15	3007[a]	3248[95]
dmu42	20	15	3172[a]	3390[95]
dmu43	20	15	3292[a]	3441[b]
dmu44	20	15	3283[a]	3488[56]
dmu45	20	15	3001[a]	3272[b]
dmu46	20	20	3575[a]	4035[b]
dmu47	20	20	3522[a]	3939[56]
dmu48	20	20	3447[a]	3763[b]
dmu49	20	20	3403[a]	3710[95]
dmu50	20	20	3496[a]	3729[95]
dmu51	30	15	3917[a]	4167[95]
dmu52	30	15	4065[a]	4311[95]
dmu53	30	15	4141[a]	4394[95]
dmu54	30	15	4202[a]	4362[b]
dmu55	30	15	4140[a]	4271[95]
dmu56	30	20	4554[a]	4941[95]
dmu57	30	20	4302[a]	4655[b]
dmu58	30	20	4319[a]	4708[95]
dmu59	30	20	4217[a]	4624[95]
dmu60	30	20	4319[a]	4755[95]
dmu61	40	15	4917[a]	5172[b]
dmu62	40	15	5033[a]	5265[b]
dmu63	40	15	5111[a]	5326[95]
dmu64	40	15	5130[36,37,38]	5250[b]
dmu65	40	15	5105[a]	5190[b]
dmu66	40	20	5391[a]	5717[95]
dmu67	40	20	5589[a]	5813[b]
dmu68	40	20	5426[a]	5773[95]
dmu69	40	20	5423[a]	5709[95]
dmu70	40	20	5501[a]	5889[b]

[36] Demirkol, Mehta, and Uzsoy (1996)
[37] Demirkol, Mehta, and Uzsoy (1997)
[38] Demirkol, Mehta, and Uzsoy (1998)
[56] Gonçalves and Resende (2014)
[69] Jain (1998)
[95] Peng, Lü, and Cheng (2015)
[a] Gharbi and Labidi (2011) using algorithms described in [51] [see 108]
[b] O. V. Shylo (2013) [see 108]

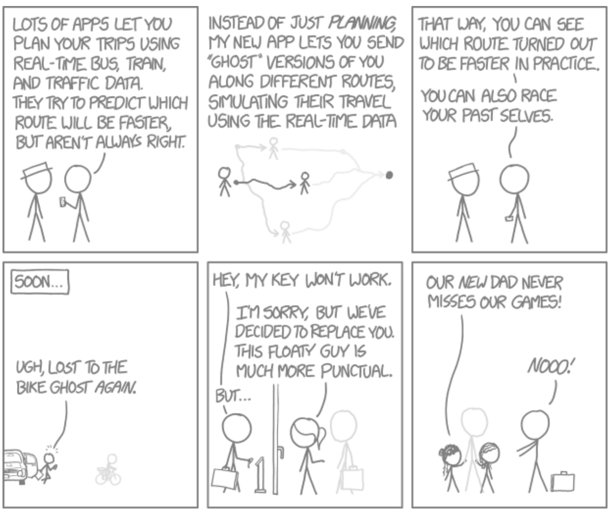
Table B.8: *Instances of Demirkol, Mehta, and Uzsoy [38] (continued)*

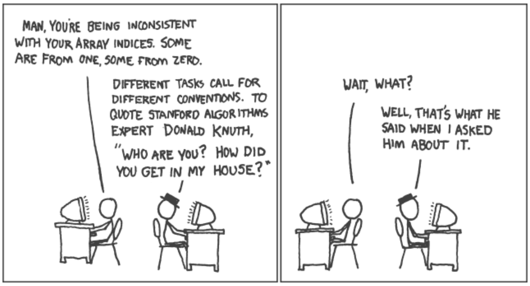
B

Instance	# jobs	# machines	Lower bound	Upper bound
dmu71	50	15	6080[a]	6223[95]
dmu72	50	15	6395[a]	6483[95]
dmu73	50	15	6001[a]	6163[95]
dmu74	50	15	6123[a]	6220[b]
dmu75	50	15	6029[a]	6197[95]
dmu76	50	20	6342[a]	6813[95]
dmu77	50	20	6499[a]	6822[95]
dmu78	50	20	6586[a]	6770[95]
dmu79	50	20	6650[a]	6970[95]
dmu80	50	20	6459[a]	6686[95]

[95] Peng, Lü, and Cheng (2015)
[a] Gharbi and Labidi (2011) using algorithms described in [51] [see 108]
[b] O. V. Shylo (2013) [see 108]

Table B.8: *Instances of Demirkol, Mehta, and Uzsoy [38] (continued)*





Glossary of Notation

Acronyms

CVRP	Capacitated Vehicle Routing Problem	30
DP	Dynamic Programming	5
GTR	Giant-Tour Representation	30
JPS	Jackson's preemptive schedule	84
JPSM	Jackson's preemptive schedule with scheduled Maintenances	106
JSSP	Job Shop Scheduling Problem	33
JSSPM	Job Shop Scheduling Problem with scheduled Maintenances	97
LP	Linear Programming	67
MC-VRP	Multiple Compartment Vehicle Routing Problem	75
MIP	Mixed-Integer Programming	97
mTSP	Multiple Traveling Salesman Problem	29
TSP	Traveling Salesman Problem	28
TSPTW	Traveling Salesman Problem with Time Windows	59
VRP	Vehicle Routing Problem	29
VRPPD	Vehicle Routing Problem with Pickup and Delivery	76
VRPTW	Vehicle Routing Problem with Time Windows	78

Common symbols

$\mathcal{O}()$	Big O notation	7
$\mathcal{LB}$	Lower bound	51
$\mathcal{UB}$	Upper bound	51
$\mathbb{N}$	Natural numbers $\{1, 2, 3, \ldots\}$	34
$\mathbb{N}_0$	Natural numbers including 0 $\{0, 1, 2, 3, \ldots\}$	23

Dynamic Programming specific symbols

β	Bookkeeping variables in a state definition	42
E	Number of expansions for a partial solution	60
H	Number of solutions to be expanded from each stage	61
$\doteq$	The first and second solution/values are equal	14
$\dot{\geq}$	The first solution/values dominates the second	14
$\dot{\neq}$	The first and second solution/values do not dominate each other	14
$\boxdot\!\!\Rightarrow$	Denotes the expansion of a solution with a node ($\varsigma \boxdot\!\!\Rightarrow i$)	6
γ	Compare variables in a state definition	13
ϕ	Fixed variables in a state definition	6
$\prec$	Precedence relation between two DP nodes	70
Ω	Set identifiers of optimal solutions	54
ς	A solution	6
$\mathring{\varsigma}$	An optimal solution	12
$\wr$	Splits the variables of ϕ and γ in a state definition	14
$\wr$	Splits the variables of γ and β in a state definition	42
ξ	State	6
$\check{\xi}$	Optimal solution of a state	6
$\hat{\xi}$	Non-dominated solutions of a state	14

Traveling Salesman Problem symbols

n	Number of nodes of a TSP problem	28
s	Start node of a TSP used in DP	28

Vehicle Routing Problem symbols

d	Destination of a vehicle	29
o	Origin of a vehicle	29
r	Request	29
v	Vehicle	29
n	Number of customer requests	29
m	Number of vehicles	29
D	Set of destinations	29
O	Set of origins	29
R	Set of requests	29
V	Set of vehicles	29

Job Shop Scheduling Problem symbols

C_o	Finish time of operation o	34
j	Job	34
m	Machine	34
$C_{\max}$	Makespan	34
$p_{\max}$	Maximum operation time	47
o	Operation	34
$\pi_j(i)$	i-th machine job j has to visit	34
p_o	Processing time of operation o	34
ψ	Schedule	34
$\alpha(\varsigma, o)$	Aptitude, earliest possible completion of o in any expansion of ς	38
$j(o)$	Job for operation o	34
$m(o)$	Machine for operation o	34
$\lambda(S)$	Last operation in S for each job	37
$\varepsilon(S)$	Next operation not in S for each job	37
$p(o)$	Processing time for operation o	34
$\eta(\varsigma)$	Possible expansions of ς	37
$\Lambda(\varsigma)$	Last operation in the sequence of ς	37
N	Number of jobs	33
M	Number of machines	33
$\mathcal{J}$	Set of jobs	34
$\mathcal{M}$	Set of machines	34
$\mathcal{O}$	Set of operations	34
$\vec{\alpha}$	Array of aptitude values	38
$\vec{\eta}$	Array of possible expansions	39

JSSP with scheduled Maintenances symbols

D	Downtime, processing time of maintenances	97
$\vec{u}$	Array of left uptime	102
u	Left uptime, before maintenance is required	102
R	Maintenance	98
U	Uptime, processing time without maintenance	97
$\mathcal{N}$	Number of maintenances	104
$\mathcal{R}$	Set of maintenances	101

JSSP bounding symbols

o^*	Current operation	112
$h^{\max}$	First possible start of the next maintenance	107

$h^{\min}$	First possible end of the previous maintenance	107
r_o	Head of operation o	84
$\tilde{r}_o$	Temporary head of operation o	108
p_o^+	Remaining processing time of operation o	84
q_o	Tail of operation o	84
K^*	Maximal set satisfying inequality (6.3)	107
t	Current time	84
$t^{\mathbf{req}}$	Next relevant time	109
$\mathcal{A}$	Set of available operations	111
$\mathcal{D}$	Set of delayed operations	111
$\mathcal{U}$	Set of unavailable operations	111
$\mathcal{M}^+$	Set of machines with operations remaining	109
$\bar{I}$	Set of all operations	104
I	Set of job operations	84
$\breve{I}$	Set of maintenance operations	104

Bibliography

[1] T.S. Abdul-Razaq, C.N. Potts, and L.N. Van Wassenhove. *A survey of algorithms for the single machine total weighted tardiness scheduling problem.* Discrete Applied Mathematics, 26.2-3: *235–253*, 1990. doi: 10.1016/0166-218X(90)90103-J (*cited on p.* 17)

[2] Joseph Adams, Egon Balas, and Daniel Zawack. *The Shifting Bottleneck Procedure for Job Shop Scheduling.* Management Science, 34.3: *391–401*, 1988. doi: 10.1287/mnsc.34.3.391 jstor: 2632051 (*cited on pp.* 85, 160–162)

[3] Ian Anderson. *Combinatorics of Finite Sets.* Dover Publications, 1987. isbn: 978-0-486-42257-2 (*cited on p.* 48)

[4] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2007. isbn: 978-0-691-12993-8 (*cited on p.* 28)

[5] David L. Applegate, Robert E. Bixby, Vašek Chvátal, William Cook, Daniel G. Espinoza, Marcos Goycoolea, and Keld Helsgaun. *Certification of an optimal TSP tour through 85,900 cities.* Operations Research Letters, 37.1: *11–15*, 2009. doi: 10.1016/j.orl.2008.09.006 (*cited on p.* 28)

[6] David L. Applegate, William J. Cook, Sanjeeb Dash, and David S. Johnson. *A Practical Guide to Discrete Optimization (Chapter 7: Dynamic Programming).* Dec. 29, 2014. url: http://www.math.uwaterloo.ca/~bico/papers/comp_chapterDP.pdf. In preparation (*cited on p.* 13)

[7] David Applegate and William Cook. *A Computational Study of the Job-Shop Scheduling Problem.* ORSA Journal on Computing, 3.2: *149–156*, 1991. doi: 10.1287/ijoc.3.2.149 (*cited on pp.* 85, 98, 160–162)

[8] David Applegate, William Cook, and André Rohe. *Chained Lin-Kernighan for Large Traveling Salesman Problems.* INFORMS Journal on Computing, 15.1: *82–92*, 2003. doi: 10.1287/ijoc.15.1.82.15157 (*cited on p.* 28)

[9] Philippe Augerat. *Approche Polyèdrale du Problème de Tournées de Véhicules.* PhD thesis. Institut National Polytechnique de Grenoble, 1995. url: http://tel.archives-ouvertes.fr/tel-00005026 (*cited on p.* 67)

[10] Egon Balas. *Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm.* Operations Research, 17.6: *941–957*, 1969. doi: 10.1287/opre.17.6.941 jstor: 168317 (*cited on p.* 160)

[11] Egon Balas and Alkis Vazacopoulos. *Guided Local Search with Shifting Bottleneck for Job Shop Scheduling.* Tech. rep. Management Science Research Report, 609. Carnegie Mellon University, 1994. (*cited on pp.* 163–166)

[12] R. Baldacci, E. Hadjiconstantinou, and A. Mingozzi. *An Exact Algorithm for the Capacitated Vehicle Routing Problem Based on a Two-Commodity Network Flow Formulation.* Operations Research, 52.5: *723–738*, 2004. doi: 10.1287/opre.1040.0111 jstor: 30036622 (*cited on p.* 67)

[13] J. E. Beasley. *OR-Library (jobshop1.txt).* url: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt (*cited on pp.* 159, 185)

[14] J. Christopher Beck, T. K. Feng, and Jean-Paul Watson. *Combining Constraint Programming and Local Search for Job-Shop Scheduling.* INFORMS Journal on Computing, 23.1: *1–14*, 2011. doi: 10.1287/ijoc.1100.0388 (*cited on p.* 165)

[15] Tolga Bektas. *The multiple traveling salesman problem: an overview of formulations and solution procedures.* Omega, 34.3: *209–219*, 2006. doi: 10.1016/j.omega.2004.10.004 (*cited on p.* 29)

[16] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957. isbn: 978-0-691-07951-6 (*cited on p.* 5)

[17] Richard Bellman. *Dynamic Programming Treatment of the Travelling Salesman Problem.* Journal of the Association for Computing Machinery, 9.1: *61–63*, 1962. doi: 10.1145/321105.321111 (*cited on pp.* 1, 2, 5, 17, 28, 187)

[18] L. Bergroth, H. Hakonen, and T. Raita. *A survey of longest common subsequence algorithms.* In: *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*: *39–48.* IEEE Computer Society, 2000. isbn: 978-0-7695-0746-0 doi: 10.1109/SPIRE.2000.878178 (*cited on p.* 8)

[19] R. Bisiani. *Beam Search.* In: *Encyclopedia of Artificial Intelligence*: *56–58.* ed. by S. C. Shapiro. Wiley & Sons, 1987. isbn: 978-0-471-50305-7 (*cited on p.* 61)

[20] Wolfgang Brinkkötter and Peter Brucker. *Solving Open Benchmark Problems for the Job Shop Problem.* Tech. rep. Osnabrücker Schriften zur Mathematik, Reihe P, 212. Universität Osnabrück, 1999. (*cited on pp.* 84, 104, 109)

[21] Wolfgang Brinkkötter and Peter Brucker. *Solving open benchmark instances for the job-shop problem by parallel head-tail adjustments.* Journal of Scheduling, 4.1: *53–64*, 2001. doi: 10.1002/1099-1425(200101/02)4:1<53::AID-JOS59>3.0.CO;2-Y (*cited on pp.* 83, 84, 104, 162–164, 167)

[22] Peter Brucker. *Scheduling Algorithms.* Springer, 2007. isbn: 978-3-540-69515-8 doi: 10.1007/978-3-540-69516-5 (*cited on p.* 33)

[23] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems.* Revised Reprint. Society for Industrial and Applied Mathematics, 2012. isbn: 978-1-61197-222-1 doi: 10.1137/1.9781611972238 (*cited on p.* 17)

[24] J. Carlier and E. Pinson. *An Algorithm for Solving the Job-shop Problem.* Management Science, 35.2: *164–176*, 1989. doi: 10.1287/mnsc.35.2.164 jstor: 2631909 (*cited on pp.* 84, 87, 160)

[25] J. Carlier and E. Pinson. *A Practical Use of Jackson's Preemptive Schedule for Solving the Job Shop Problem.* Annals of Operations Research, 26: *269–287*, 1990. (*cited on p.* 161)

[26] J. Carlier and E. Pinson. *Adjustment of heads and tails for the job-shop problem.* European Journal of Operational Research, 78.2: *146–161*, 1994. doi: 10.1016/0377-2217(94)90379-4 (*cited on p.* 161)

[27] Robert L. Carraway and Robert L. Schmidt. *An Improved Discrete Dynamic Programming Algorithm for Allocating Resources among Interdependent Projects.* Management Science, 37.9: *1195–1200*, 1991. doi: 10.1287/mnsc.37.9.1195 jstor: 2632334 (*cited on p.* 51)

[28] Emmanual D. Chajakis and Monique Guignard. *Scheduling Deliveries in Vehicles with Multiple Compartments.* Journal of Global Optimization, 26.1: *43–78*, 2003. doi: 10.1023/A:1023067016014 (*cited on p.* 75)

[29] T.C.E. Cheng, Bo Peng, and Zhipeng Lü. *A hybrid evolutionary algorithm to solve the job shop scheduling problem.* Annals of Operations Research: *1–15*, 2013. doi: 10.1007/s10479-013-1332-5 (*cited on p.* 163)

[30] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu, eds. *Scheduling theory and its applications.* John Wiley & Sons, 1995. isbn: 978-0-471-94059-3 (*cited on p.* 33)

[31] N. Christofides and S. Eilon. *An Algorithm for the Vehicle-Dispatching Problem.* Journal of the Operational Research Society, 20.3: *309–318*, 1969. doi: 10.1057/jors.1969.75 jstor: 3008733 (*cited on p.* 67)

[32] N. Christofides, A. Mingozzi, and P. Toth. *The vehicle routing problem.* In: *Combinatorial Optimization*, 11: *315–338*. ed. by N. Christofides, A. Mingozzi, P. Toth, and C. Sandi. Wiley & Sons, 1979. isbn: 978-0-471-99749-8 (*cited on p.* 67)

[33] Nicos Christofides, A. Mingozzi, and P. Toth. *State-Space Relaxation Procedures for the Computation of Bounds to Routing Problems.* Networks, 11.2: *145–164*, 1981. doi: 10.1002/net.3230110207 (*cited on p.* 59)

[34] William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation.* Princeton University Press, 2012. isbn: 978-0-691-15270-7 (*cited on p.* 28)

[35] Mauro Dell'Amico, Giovanni Righini, and Matteo Salani. *A Branch-and-Price Approach to the Vehicle Routing Problem with Simultaneous Distribution and Collection.* Transportation Science, 40.2: *235–247*, 2006. doi: 10.1287/trsc.1050.0118 jstor: 25769299 (*cited on p.* 82)

[36] Ebru Demirkol, Sanjay V. Mehta, and Reha Uzsoy. *Benchmarking for Shop Scheduling Problems.* Tech. rep. Research memorandum, 96-4. Purdue University, 1996. (*cited on pp.* 168, 169)

[37] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. *A Computational Study of Shifting Bottleneck Procedures for Shop Scheduling Problems.* Journal of Heuristics, 3.2: *111–137*, 1997. doi: 10.1023/A:1009627429878 (*cited on pp.* 168, 169)

[38] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. *Benchmarks for shop scheduling problems.* European Journal of Operational Research, 109.1: *137–141*, 1998. doi: 10.1016/S0377-2217(97)00019-2 (*cited on pp.* 85, 167–170)

[39] S. E. Dreyfus and R. A. Wagner. *The steiner problem in graphs.* Networks, 1.3: *195–207*, 1971. doi: 10.1002/net.3230010302 (*cited on pp.* 17, 20)

[40] Stuart E. Dreyfus and Averill M. Law. *The Art and Theory of Dynamic Programming.* Academic Press, 1977. isbn: 978-0-08-095639-8 (*cited on p.* 72)

[41] M.E. Dyer, W.O. Riha, and J. Walker. *A Hybrid Dynamic Programming/Branch-and-bound Algorithm for the Multiple-choice Knapsack Problem.* Journal of Computational and Applied Mathematics, 58.1: *43–54*, 1995. doi: 10.1016/0377-0427(93)E0264-M (*cited on p.* 51)

[42] Abdellah El Fallahi, Christian Prins, and Roberto Wolfler Calvo. *A memetic algorithm and a tabu search for the multi-compartment vehicle routing problem.* Computers & Operations Research, 35.5: *1725–1741*, 2008. doi: 10.1016/j.cor.2006.10.006 (*cited on p.* 75)

[43] Dominique Feillet. *A tutorial on column generation and branch-and-price for vehicle routing problems.* 4OR, 8.4: *407–424*, 2010. doi: 10.1007/s10288-010-0130-z (*cited on p.* 82)

[44] H. Fisher and G. L. Thompson. *Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules.* In: *Industrial Scheduling*, 15: *225–251*. ed. by J. F. Muth and G. L. Thompson. Prentice Hall, 1963. oclc: 781815542 (*cited on pp.* 85, 87, 160)

[45] Marshall L. Fisher. *Optimal Solution of Vehicle Routing Problems Using Minimum K-Trees*. Operations Research, 42.4: *626–642*, 1994. doi: 10.1287/opre.42.4.626 jstor: 171617 (*cited on p.* 67)

[46] M. Florian, P. Trepant, and G. McMahon. *An Implicit Enumeration Algorithm for the Machine Sequencing Problem*. Management Science, 17.12: *B-782–B-792*, 1971. doi: 10.1287/mnsc.17.12.B782 jstor: 2629469 (*cited on p.* 160)

[47] U. Fößmeier and M. Kaufmann. *On Exact Solutions for the Rectilinear Steiner Tree Problem Part I: Theoretical Results*. Algorithmica, 26.1: *68–99*, 2000. doi: 10.1007/s004539910005 (*cited on p.* 17)

[48] B. Fuchs, W. Kern, D. Molle, S. Richter, P. Rossmanith, and X. Wang. *Dynamic Programming for Minimum Steiner Trees*. Theory of Computing Systems, 41.3: *493–500*, 2007. doi: 10.1007/s00224-007-1324-4 (*cited on p.* 17)

[49] Birger Funke, Tore Grünert, and Stefan Irnich. *Local Search for Vehicle Routing and Scheduling Problems: Review and Conceptual Integration*. Journal of Heuristics, 11.4: *267–306*, 2005. doi: 10.1007/s10732-005-1997-2 (*cited on p.* 30)

[50] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979. isbn: 978-0-7167-1045-5 (*cited on p.* 17)

[51] Anis Gharbi and Mohamed Labidi. *Extending the Single Machine-Based Relaxation Scheme for the Job Shop Scheduling Problem*. Electronic Notes in Discrete Mathematics, 36: *1057–1064*, 2010. doi: 10.1016/j.endm.2010.05.134 (*cited on pp.* 165–170)

[52] Billy E. Gillett and Jerry G. Johnson. *Multi-terminal vehicle-dispatch algorithm*. Omega, 4.6: *711–718*, 1976. doi: 10.1016/0305-0483(76)90097-9 (*cited on p.* 67)

[53] Asvin Goel. *Vehicle Scheduling and Routing with Drivers' Working Hours*. Transportation Science, 43.1: *17–26*, 2009. doi: 10.1287/trsc.1070.0226 jstor: 25769429 (*cited on p.* 78)

[54] Asvin Goel. *Truck Driver Scheduling in the European Union*. Transportation Science, 44.4: *429–441*, 2010. doi: 10.1287/trsc.1100.0330 (*cited on pp.* 78, 79)

[55] Asvin Goel and Leendert Kok. *Truck Driver Scheduling in the United States*. Transportation Science, 46.3: *317–326*, 2012. doi: 10.1287/trsc.1110.0382 (*cited on p.* 78)

[56] José Fernando Gonçalves and Mauricio G. C. Resende. *An extended Akers graphical method with a biased random-key genetic algorithm for job-shop scheduling*. International Transactions in Operational Research, 21.2: *215–246*, 2014. doi: 10.1111/itor.12044 (*cited on pp.* 163–166, 168, 169)

[57] Samuel Gorenstein. *Printing Press Scheduling for Multi-Edition Periodicals*. Management Science, 16.6: *B-373–B-383*, 1970. doi: 10.1287/mnsc.16.6.B373 jstor: 2628724 (*cited on p.* 30)

[58] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. *Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey*. In: *Discrete Optimization II*: *287–326*. ed. by E.L. Johnson P.L. Hammer and B.H. Korte. Annals of Discrete Mathematics, 5. Elsevier, 1979. isbn: 978-0-08-086767-0 doi: 10.1016/S0167-5060(08)70356-X (*cited on pp.* 17, 33)

[59] J. Gromicho, J.J. van Hoorn, A.L. Kok, and J.M.J. Schutten. *Restricted dynamic programming: A flexible framework for solving realistic VRPs*. Computers & Operations Research, 39.5: *902–909*, 2012. doi: 10.1016/j.cor.2011.07.002 (*cited on pp.* 2, 27, 188)

[60] Joaquim A.S. Gromicho, Jelke J. van Hoorn, Francisco Saldanha-da-Gama, and Gerrit T. Timmer. *Solving the Job-shop Scheduling Problem Optimally by Dynamic Programming*. Computers & Operations Research, 39.12: *2968–2977*, 2012. doi: 10.1016/j.cor.2012.02.024 (*cited on pp.* 2, 27, 187)

[61] Gregory Gutin and Abraham P. Punnen, eds. *Traveling Salesman Problem and Its Variations*. Combinatorial Optimization, 12. Springer, 2002. isbn: 978-0-387-44459-8 doi: 10.1007/b101971 (*cited on p.* 28)

[62] Michael Held and Richard M. Karp. *A Dynamic Programming Approach to Sequencing Problems*. Journal of the Society for Industrial and Applied Mathematics, 10.1: *196–210*, 1962. doi: 10.1137/0110015 (*cited on pp.* 1, 2, 5, 17, 28, 187)

[63] Keld Helsgaun. *An effective implementation of the Lin-Kernighan traveling salesman heuristic*. European Journal of Operational Research, 126.1: *106–130*, 2000. doi: 10.1016/S0377-2217(99)00284-2 (*cited on p.* 28)

[64] Keld Helsgaun. *An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic*. Tech. rep. Datalogiske skrifter, 109. Roskilde University, 2006. url: http://www.akira.ruc.dk/~keld/research/LKH/ (*cited on p.* 28)

[65] Keld Helsgaun. *General k-opt submoves for the Lin-Kernighan TSP heuristic*. Mathematical Programming Computation, 1.2–3: *119–163*, 2009. doi: 10.1007/s12532-009-0004-6 (*cited on p.* 28)

[66] André Henning. *Praktische Job-Shop Scheduling-Probleme*. PhD thesis. Friedrich-Schiller-Universität Jena, 2002. url: http://www.db-thueringen.de/servlets/DocumentServlet?id=873 (*cited on pp.* 165, 167)

[67] J. J. van Hoorn. *Job Shop Instances and Solutions*. url: http://jobshop.jjvh.nl (*cited on pp.* 96, 159)

[68] Lawrence Hubert, Phipps Arabie, and Jacqueline Meulman. *Combinatorial Data Analysis: Optimization by Dynamic Programming*. Society for Industrial and Applied Mathematics, 2001. isbn: 978-0-89871-478-4 (*cited on p.* 17)

[69] Anant Singh Jain. *A Multi-Level Hybrid Framework for the Deterministic Job-Shop Scheduling Problem*. PhD thesis. University Of Dundee, 1998. url: `http://www.personal.dundee.ac.uk/~asjain/papers/publications.html` (*cited on pp.* 166, 168, 169)

[70] A.S. Jain and S. Meeran. *Deterministic job-shop scheduling: Past, present and future*. European Journal of Operational Research, 113.2: *390–434*, 1999. doi: `10.1016/S0377-2217(98)00113-1` (*cited on pp.* 159, 162)

[71] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004. isbn: 978-3-540-40286-2 doi: `10.1007/978-3-540-24777-7` (*cited on p.* 10)

[72] Leendert Kok. *Congestion avoidance and break scheduling within vehicle routing*. PhD thesis. University of Twente, 2010. isbn: 978-90-365-2990-7. doi: `10.3990/1.9789036529907` (*cited on p.* 78)

[73] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. *Solving Open Job-Shop Scheduling Problems by SAT Encoding*. IEICE Transactions on Information and Systems, E93.D.8: *2316–2318*, 2010. doi: `10.1587/transinf.E93.D.2316` (*cited on pp.* 162, 164)

[74] H. W. Kuhn. *The Hungarian method for the assignment problem*. Naval Research Logistics Quarterly, 2.1-2: *83–97*, 1955. doi: `10.1002/nav.3800020109` (*cited on p.* 17)

[75] Peter J. M. van Laarhoven, Emile H. L. Aarts, and Jan Karel Lenstra. *Job shop scheduling by simulated annealing*. Operations Research, 40.1: *113–125*, 1992. doi: `10.1287/opre.40.1.113` jstor: `171189` (*cited on p.* 160)

[76] Gilbert Laporte. *The vehicle routing problem: An overview of exact and approximate algorithms*. European Journal of Operational Research, 59.3: *345–358*, 1992. doi: `10.1016/0377-2217(92)90192-C` (*cited on p.* 29)

[77] Gilbert Laporte. *Fifty Years of Vehicle Routing*. Transportation Science, 43.4: *408–416*, 2009. doi: `10.1287/trsc.1090.0301` (*cited on p.* 29)

[78] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985. isbn: 978-0-471-90413-7 (*cited on p.* 28)

[79] S. Lawrence. *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. Carnegie-Mellon University, 1984. (*cited on pp.* 85, 160, 161)

[80] Édouard Lucas. *Théorie des nombres.* 1: *Le calcul des nombres entiers. Le calcul des nombres rationnels. La divisibilité arithmétique.* Gauthier-Villars et Fils, 1891. OCLC: 490285600 (*cited on p.* 6)

[81] Chryssi Malandraki and Robert B. Dial. *A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem.* European Journal of Operational Research, 90.1: *45–55*, 1996. DOI: 10.1016/0377-2217(94)00299-1 (*cited on p.* 61)

[82] Roy E. Marsten and Thomas L. Morin. *A hybrid approach to discrete mathematical programming.* Mathematical Programming, 14.1: *21–40*, 1978. DOI: 10.1007/BF01588949 (*cited on p.* 51)

[83] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, 1990. ISBN: 978-0-471-92420-3 URL: http://www.or.deis.unibo.it/knapsack.html (*cited on pp.* 10, 105)

[84] Paul Douglas Martin. *A time-oriented approach to computing optimal schedules for the job-shop scheduling problem.* PhD thesis. Cornell University, 1996. OCLC: 64683112 (*cited on pp.* 161–163)

[85] Hirofumi Matsuo, Chang Juck Suh, and Robert S. Sullivan. *A Controlled Search Simulated Annealing Method for the General Job-Shop Scheduling Problem.* Working paper 03-04-88. The University of Texas at Austin, 1988. (*cited on pp.* 160, 161)

[86] Graham McMahon and Michael Florian. *On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness.* Operations Research, 23.3: *475–482*, 1975. DOI: 10.1287/opre.23.3.475 JSTOR: 169697 (*cited on p.* 160)

[87] Randall Munroe. *xkcd: A webcomic of romance, sarcasm, math, and language.* URL: http://xkcd.com (*cited on p.* 193)

[88] Yuichi Nagata and Isao Ono. *Guided Constructive Local Search for the Job Shop Scheduling Problem.* 2013. submitted (*cited on p.* 166)

[89] Yuichi Nagata and Satoshi Tojo. *Guided Ejection Search for the Job Shop Scheduling Problem.* In: *Evolutionary Computation in Combinatorial Optimization*: *168–179*. ed. by Carlos Cotta and Peter Cowling. LNCS, 5482. Springer, 2009. ISBN: 978-3-642-01008-8 DOI: 10.1007/978-3-642-01009-5_15 (*cited on p.* 163)

[90] Eugeniusz Nowicki and Czeslaw Smutnicki. *A Fast Taboo Search Algorithm for the Job Shop Problem.* Management Science, 42.6: *797–813*, 1996. DOI: 10.1287/mnsc.42.6.797 JSTOR: 2634595 (*cited on pp.* 161, 164–167)

[91] Eugeniusz Nowicki and Czeslaw Smutnicki. *Some new tools to solve the job-shop problem.* Tech. rep. 60/02. Wroclaw University of Technology, 2001. (*cited on p.* 168)

[92] Eugeniusz Nowicki and Czeslaw Smutnicki. *An Advanced Tabu Search Algorithm for the Job Shop Problem.* Journal of Scheduling, 8.2: *145–159*, 2005. doi: 10.1007/s10951-005-6364-5 (*cited on pp.* 163–166)

[93] Panos M. Pardalos and Oleg V. Shylo. *An Algorithm for the Job Shop Scheduling Problem based on Global Equilibrium Search Techniques.* Computational Management Science, 3.4: *331–348*, 2006. doi: 10.1007/s10287-006-0023-y (*cited on pp.* 165, 167)

[94] Panos M. Pardalos, Oleg V. Shylo, and Alkis Vazacopoulos. *Solving job shop scheduling problems utilizing the properties of backbone and "big valley".* Computational Optimization and Applications, 47.1: *61–76*, 2010. doi: 10.1007/s10589-008-9206-5 (*cited on pp.* 165, 167)

[95] Bo Peng, Zhipeng Lü, and T. C. E. Cheng. *A tabu search/path relinking algorithm to solve the job shop scheduling problem.* Computers & Operations Research, 53: *154–164*, 2015. doi: 10.1016/j.cor.2014.08.006 (*cited on pp.* 163, 165, 166, 168–170)

[96] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer, 2016. isbn: 978-3-319-26578-0 doi: 10.1007/978-3-319-26580-3 (*cited on p.* 33)

[97] Leonardo Pisano. *Liber Abaci.* 1202. (*cited on p.* 6)

[98] R. C. Prim. *Shortest connection networks and some generalizations.* Bell System Technology Journal, 36.6: *1389–1401*, 1957. doi: 10.1002/j.1538-7305.1957.tb01515.x (*cited on p.* 21)

[99] Jakob Puchinger and Peter J. Stuckey. *Automating Branch-and-bound for Dynamic Programs.* In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation: 81–89.* ACM, 2008. isbn: 978-1-59593-977-7 doi: 10.1145/1328408.1328421 (*cited on p.* 51)

[100] X. Qi, T. Chen, and F. Tu. *Scheduling the Maintenance on a Single Machine.* Journal of the Operational Research Society, 50.10: *1071–1078*, 1999. doi: 10.1057/palgrave.jors.2600791 jstor: 3009932 (*cited on p.* 97)

[101] T. Ralphs. *Vehicle Routing Data Sets.* 2003. url: http://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/index.htm (*cited on p.* 67)

[102] César Rego and Fred Glover. *Local search and metaheuristics.* In: *Traveling Salesman Problem and Its Variations*, 8: *309–368.* ed. by Gregory Gutin and Abraham P. Punnen. Combinatorial Optimization, 12. Springer, 2002. isbn: 978-0-387-44459-8 doi: 10.1007/0-306-48213-4_8 (*cited on p.* 61)

[103] Gerhard Reinelt. *TSPLIB–A Traveling Salesman Problem Library.* ORSA Journal on Computing, 3.4: *376–384*, 1991. doi: 10.1287/ijoc.3.4.376 (*cited on p.* 67)

[104] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. LNCS, 840. Springer-Verlag, 1994. ISBN: 978-3-540-58334-9 DOI: 10.1007/3-540-48661-5 (*cited on p.* 28)

[105] Panagiotis P. Repoussis, Christos D. Tarantilis, and George Ioannou. *A Hybrid Metaheuristic for a Real Life Vehicle Routing Problem*. In: *Numerical Methods and Applications*: *247–254*. ed. by Todor Boyanov, Stefka Dimova, Krassimir Georgiev, and Geno Nikolov. LNCS, 4310. Springer, 2007. ISBN: 978-3-540-70940-4 DOI: 10.1007/978-3-540-70942-8_29 (*cited on p.* 75)

[106] Linus Schrage. *Solving Resource-Constrained Network Problems by Implicit Enumeration-Nonpreemptive Case*. Operations Research, 18.2: *263–278*, 1970. DOI: 10.1287/opre.18.2.263 JSTOR: 168683 (*cited on p.* 160)

[107] Linus Schrage and Kenneth R. Baker. *Dynamic Programming Solution of Sequencing Problems with Precedence Constraints*. Operations Research, 26.3: *444–449*, 1978. DOI: 10.1287/opre.26.3.444 JSTOR: 169755 (*cited on p.* 24)

[108] Oleg V. Shylo. *Job Shop Scheduling at Oleg V. Shylo: Personal Webpage*. URL: http://optimizizer.com/jobshop.php (*cited on pp.* 159, 163, 165–170)

[109] Laurence Sigler. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Springer, 2002. ISBN: 978-0-387-40737-1 DOI: 10.1007/978-1-4613-0079-3 (*cited on p.* 6)

[110] Robert H. Storer, S. David Wu, and Renzo Vaccari. *New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling*. Management Science, 38.10: *1495–1509*, 1992. DOI: 10.1287/mnsc.38.10.1495 (*cited on pp.* 85, 163)

[111] E.D. Taillard. *Benchmarks for basic scheduling problems*. European Journal of Operational Research, 64.2: *278–285*, 1993. DOI: 10.1016/0377-2217(93)90182-M (*cited on pp.* 85, 164–167)

[112] Éric D. Taillard. *Éric Taillard's page*. URL: http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html (*cited on pp.* 159, 164–166)

[113] Éric D. Taillard. *Parallel Taboo Search Techniques for the Job Shop Scheduling Problem*. ORSA Journal on Computing, 6.2: *108–117*, 1994. DOI: 10.1287/ijoc.6.2.108 (*cited on pp.* 164–167)

[114] Søren Thomsen. *Metaheuristikker kombineret med Branch & Bound*. MA thesis. Copenhagen Business School, 1997. OCLC: 464628711 (*cited on pp.* 163, 164)

[115] Paolo Toth and Daniele Vigo. *The Granular Tabu Search and Its Application to the Vehicle-Routing Problem*. INFORMS Journal on Computing, 15: *333–346*, 4 2003. DOI: 10.1287/ijoc.15.4.333.24890 (*cited on p.* 61)

[116] Paolo Toth and Daniele Vigo, eds. *Vehicle Routing: Problems, Methods, and Applications.* Second Edition. Society for Industrial and Applied Mathematics, 2014. isbn: 978-1-61197-358-7 doi: 10.1137/1.9781611973594 (*cited on p.* 29)

[117] R. J. M. Vaessens. *Addition to the OR-Library [13].* 1996 (*cited on p.* 163)

[118] R. J. M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. *Job Shop Scheduling by Local Search.* INFORMS Journal on Computing, 8.3: *302–317*, 1996. doi: 10.1287/ijoc.8.3.302 (*cited on p.* 161)

[119] Petr Vilím, Philippe Laborie, and Paul Shaw. *Failure-Directed Search for Constraint-Based Scheduling — Detailed Experimental Results.* url: http://vilim.eu/petr/cpaior2015-results.pdf (*cited on p.* 159)

[120] Petr Vilím, Philippe Laborie, and Paul Shaw. *Failure-Directed Search for Constraint-Based Scheduling.* In: *Integration of AI and OR Techniques in Constraint Programming*: *437–453.* ed. by Laurent Michel. LNCS, 9075. Springer, 2015. isbn: 978-3-319-18007-6 doi: 10.1007/978-3-319-18008-3_30 (*cited on p.* 159)

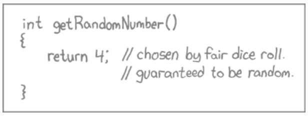
[121] Robert A. Wagner and Michael J. Fischer. *The String-to-String Correction Problem.* Journal of the Association for Computing Machinery, 21.1: *168–173*, 1957. doi: 10.1145/321796.321811 (*cited on p.* 8)

[122] Gerhard J. Woeginger. *Exact Algorithms for NP-Hard Problems: A Survey.* In: *Combinatorial Optimization–Eureka, You shrink!*: *185–207.* ed. by Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. LNCS, 2570. Springer, 2003. isbn: 978-3-540-00580-3 doi: 10.1007/3-540-36478-1_17 (*cited on p.* 33)

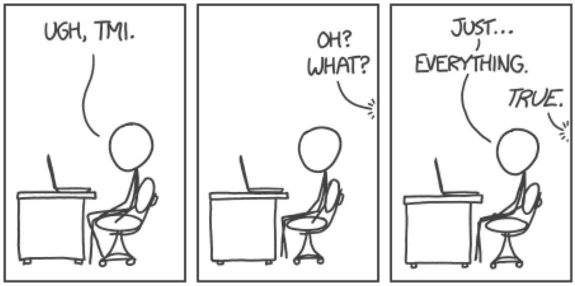
[123] Takeshi Yamada and Ryohei Nakano. *A genetic algorithm applicable to large-scale job-shop instances.* In: *Parallel instance solving from nature 2*: *281–290.* ed. by Reinhard Männer and Bernard Manderick. Elsevier, 1992. isbn: 978-0-444-89730-5 (*cited on pp.* 85, 164)

[124] ChaoYong Zhang, PeiGen Li, ZaiLin Guan, and YunQing Rao. *A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem.* Computers & Operations Research, 34.11: *3229–3242*, 2007. doi: 10.1016/j.cor.2005.12.002 (*cited on p.* 168)

[125] ChaoYong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. *A very fast TS/SA algorithm for the job shop scheduling problem.* Computers & Operations Research, 35.1: *282–294*, 2008. doi: 10.1016/j.cor.2006.02.024 (*cited on pp.* 162, 164, 165)




NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!



Summary

This dissertation examines Dynamic Programming algorithms for routing and scheduling. These algorithms are based on the famous Dynamic Programming algorithm for the Traveling Salesman Problem already described over 50 years ago by Held and Karp [62] (and also independently by Bellman [17]). This algorithm is largely viewed as theoretical. Based on this algorithm we created new algorithms for the Vehicle Routing Problem and Job Shop Scheduling Problem and variants of these problems.

For several problems such a Dynamic Programming algorithm provides the best known complexity for an algorithm that guarantees to find the optimal value to the problem. For the Traveling Salesman Problem this was already known for the Job Shop Scheduling Problem we proved this in Gromicho, van Hoorn, Saldanha-da-Gama, and Timmer [60]. Most Dynamic Programming algorithms over sets have limited practical use as their running time and memory requirements are exponential. However, we show that by the use of bounding such Dynamic Programming algorithms can become practical applicable. We also show several ways to convert such Dynamic Programming algorithms into heuristic algorithms which then, although the optimality guarantee is lost, have practical value.

The basis of these Dynamic Programming algorithms is recursion over sets. To use Dynamic Programming over sets on a problem a solution for a such problem must be represented as a specific sequence of a set of nodes. The Dynamic Programming algorithm evaluates the best sequence based on the best sequence for each subset of the nodes. The power in the Dynamic Programming algorithm lies in the fact that it enables to consider all sequences by the evaluation of sequences based on each subset. Although there are still exponentially many subsets (2^n) this is exponentially less than the number of possible sequences ($n!$).

The Dynamic Programming algorithm can be converted into an iterative process to find all optimal solutions. We show in general how to create such a procedure and for the Job Shop Scheduling Problem we show the procedure in more detail. We also show the results and the total number of optimal solutions for small Job Shop Scheduling Problem benchmark instances.

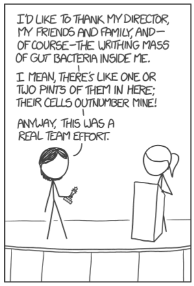
For the Vehicle Routing Problem we show how to incorporate a large number of extensions to the Vehicle Routing Problem optimally into the Dynamic Programming algorithm. We also show the effects of these extensions on the complexity of the Dynamic Programming algorithm. This creates a general

framework to solve Vehicle Routing Problems as also described in Gromicho, van Hoorn, Kok, and Schutten [59]. We also show briefly how such framework can be used as pricing instrument in a column generation technique. For the Capacitated Vehicle Routing Problem we show with computational results what the effect of bounding can be on the Dynamic Programming state space.

We describe how to create a Dynamic Programming algorithm to solve the Job Shop Scheduling Problem, which provides the best time complexity to solve this problem to optimality. We show computational results for the Dynamic Programming algorithm for the Job Shop Scheduling Problem with and without the use of bounding. For a few Job Shop Scheduling Problem benchmark instances we are able to improve the best known lower bounds.

We create a new extension to the Job Shop Scheduling Problem by adding maintenance times to the machines. For this new problem we create a Mixed-Integer Programming formulation as well as a Dynamic Programming algorithm. We also create a bounding algorithm to be used within this Dynamic Programming algorithm. A comparison of computational results for both algorithms show that Dynamic Programming can outperform a state of the art Mixed-Integer Programming solver using this Mixed-Integer Programming formulation.

For well-known benchmark instances for the Job Shop Scheduling Problem we provide the best known values for the upper and lower bounds as well as the origin of these bounds. This information as well as detailed results of all computational experiments can be found in the appendix.



Acknowledgements

After almost eight years all my research is finally written down, with a cover this time. My PhD research was a great adventure, with amazing experiences and wonderful challenges. A lot of people contributed to the realization of this dissertation, it is a pleasure to thank those people. Without the illusion of being able to be exhaustive I would like to take this opportunity to give thanks to the following people.

Joaquim and Gerrit, my supervisors, I am grateful for the opportunity you gave me. You took me on this adventure and I think we can all be proud of the result. This was a far from typical PhD project and I think we all struggled a bit how to find our way outside of the beaten path.

I would like to thank you for all the valuable feedback you provided during our discussions. I have fond memories of our regular meetings, especially of the discussions about everything not directly related to my research. Also, I would like to thank ORTEC for the flexibility and time that enabled me to do my research and finish this dissertation.

I would like to express my sincere gratitude to the PhD committee for their comments on this dissertation. Although it was at the very end of this endeavour, their remarks considerably helped to improve the final text and structure of this dissertation.

Geert-Pieter and Sander, my paranymphs, I would like to thank you both for your support to continue the writing of this dissertation. Sander especially for reading a rudimentary version and providing comments from a different angle. Geert-Pieter foremost for all the evenings we spend together at the library, both writing, and stimulating each other to keep writing.

I also would like to thank the students I supervised, directly or more indirectly. Chagiet, Eline, Manon, Quirijn, Rutger, Steven, Stevo, and Wenda, you all contributed your own piece to my research. It was a pleasure to supervise your theses, this was the most inspiring part of my teaching roles.

I would like thank all my colleagues at ORTEC for their support and discussions. In particular I would like to thank Gerhard and Jeroen. Gerhard for the encouragements when I thought I was stuck, but also for lending me helpful books and pointing me in new directions. Jeroen for answering all my questions regarding my code implementations and allowing me to use his computer to perform test runs.

Francisco, Leendert, and Marco, co-authors of my already published papers, I would like to thank you for our joint research. Francisco, the period you visited Amsterdam was one of the most exhilarating times of my research. I am specifically thankful for your invitation to Lisbon, to finish our paper together. Karianne and I had a fantastic time in Portugal.

I am indebted to Ignacio and Agustin for finding a flaw in my work and showing me this error. I am glad we were able to correct this flaw together and I hope this is a starting point for a fruitful cooperation.

Three people I would like to thank for their contribution in me, without them I would not be who I am today. Henk and Willemien, my parents, who tried to give the right example, who provided me with good advice, both of which I would not always follow. I would like to thank you for the way how you almost always encouraged and supported me, never giving me the impression I could not succeed. Niels, my little brother, I would like to thank you for being there and standing up for me at the most crucial moments.

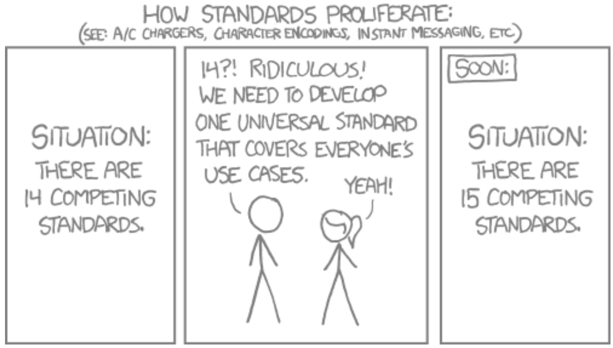
Willemien, I am grateful for your help in improving this text, your endless re-reading of new versions clearly enhanced this dissertation.

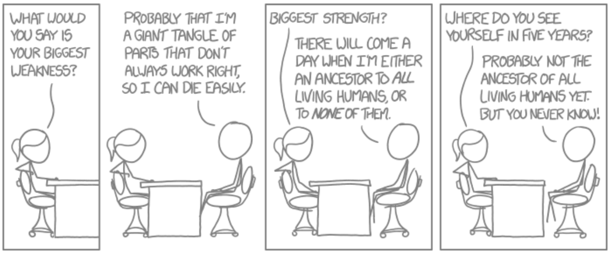
Karianne, darling, I have no way to express my gratitude for our past decade together. When we decided eight years ago that I would start this journey, we could not have imagined how different our lives would be today, when I finish my PhD project. Maybe we could have envisioned the facts but certainly not the effects. In those eight years, we bought a house, we, well especially you, made it into a home. With two beautiful, thoughtful, joyful, and just amazing kids, and even a third on the way.

Lieve Bjarne en Ilva, het schrijven is eindelijk klaar. Bedankt voor de vele fantastische uitspraken en vragen waar jullie mee kwamen tijdens het schrijven, wanneer ik met jullie thuis aan het werk was. Dit gaf vaak een welkom moment van ontspanning.

Dear Karianne, we knew my research would take up a lot of time, but we had no idea how. Thank you for putting up with me, working every night, with no room in my mind to decide on other things. Thank you for creating the space and the time for finishing this project. And above all for just being you.

JJ




WHAT WOULD YOU SAY IS YOUR BIGGEST WEAKNESS?
PROBABLY THAT I'M A GIANT TANGLE OF PARTS THAT DON'T ALWAYS WORK RIGHT, SO I CAN DIE EASILY.
BIGGEST STRENGTH?
THERE WILL COME A DAY WHEN I'M EITHER AN ANCESTOR TO ALL LIVING HUMANS, OR TO NONE OF THEM.
WHERE DO YOU SEE YOURSELF IN FIVE YEARS?
PROBABLY NOT THE ANCESTOR OF ALL LIVING HUMANS YET. BUT YOU NEVER KNOW!

HOW TO FREAK OUT A MOBILE APP USER

List of Comics

All comics in this dissertation are created by Randall Munroe and are published on xkcd [87]. I would like to thank him for allowing the use of these comics in this dissertation. Here follows a list of all comics used in this dissertation including their number, title and title text as published on xkcd.

Comics at chapter start

Comics at chapter start (continued)

Comics at chapter end

Comics at chapter end (continued)

I CAN NEVER FIGURE OUT WHAT TO SAY ABOUT BABIES.