Université catholique de Louvain

Louvain School of Management (LSM)

LOUVAIN School of Management

An Ant Colony System for solving the Job Sequencing and Tool Switching Problem

Mémoire recherche réalisé par
Henri Dehaybe

en vue de l'obtention du titre de
Master en ingénieur de gestion, à finalité spécialisée

Promoteur
Daniele Catanzaro

Année académique 2017-2018

Abstract

In this thesis, we investigate the Job Sequencing and Tool Switching Problem (SSP), a well-known NP-hard problem in operational research arising in flexible manufacturing systems. We present the problem and all its known properties and we review the existing solving procedures in the literature. Then we present a well-known class of metaheuristics, *Ant Colony Optimization* (ACO), and how it is applied to solve the Travelling Salesman Problem (TSP), a famous problem that shares some properties with the SSP.

We hybridize a customized Ant Colony System with a new local search inspired by the *2.5-opt* for the TSP. After some tuning, the proposed algorithm becomes strongly competitive with the existing state of the art methods for that problem in the literature.

I thank my thesis supervisor, Daniele Catanzaro, for all his input and for his dedicated time and patient guidance. I am truly grateful because I genuinely took pleasure in researching and writing this thesis. It was an enjoyable and challenging experience and I discovered a field I have become passionate about.

I am also thankful to my parents and Fabienne M. who accepted to proofread this work and for their valuable comments. My mother in particular for letting me run my experiments on her personal computer and use its CPU power.

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

The Job Sequencing and Tool Switching Problem (SSP) is a combinatorial optimization problem in operations research. The problem first arose in manufacturing systems where a single flexible machine was used and is still relevant to this date. In this problem, the machine must process a set of different jobs, each of them requiring a set of tools to be loaded on the machine to be performed. But the magazine of the machine has a limited capacity of tools that it can handle at the same time. Thus, it must be stopped to switch some of the tools to go on with the sequence. These numerically controlled machines require a fine tuning during the tool changes, or a washing, or anything else that makes the switching time significant compared to the job processing time. The tool switches should therefore be avoided as much as possible (Tang & Denardo, 1988a). This is precisely the objective of the SSP: it must find the optimal order, or sequence, of jobs so that the total number of tool switches is minimized when they are processed. We will see later that there exist many variants to this problem, with different objectives and/or constraints that are also relevant in flexible manufacturing systems. In the single machine SSP, the problem considered in this thesis, the only constraints are 1) the capacity of the machine and 2) that at each instant of the sequence, all the required tools are loaded to perform the next job. The free slots in the magazine can accommodate any other tool even if they are not needed, allowing to cleverly keep them for future use in the sequence as illustrated in figure 1.

The SSP is a complex problem, it is *NP-Hard*: even the best exact approaches cannot solve instances with more than 15 jobs to (guaranteed) optimality. This is why a preferred approach is the use of heuristics. Heuristics aim to find a reasonably good solution without any guarantee that it is the optimal one. They have the disadvantage of finding a single local optimum in an immense solution space. Metaheuristics are higher level strategies that guide heuristic procedures to create a process capable of escaping local optima to explore more regions of the solution space (Gendreau & Potvin, 2010). They range from

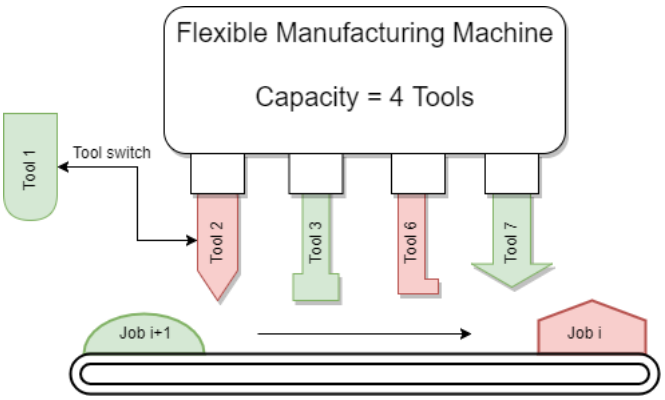


Figure 1: When job i is done, two tools required for job $i+1$ are already loaded on this flexible machine. One tool switch is needed to load the third one.

simple random perturbations to algorithms that imitate the evolution and the natural selection mechanism to create stronger individuals.

The objective of this thesis is to implement a famous class of metaheuristics called Ant Colony Optimization (ACO) to assess how well it solves the SSP. Ant Colony algorithms are inspired by the behaviour of real ants when they are foraging for food. They leave a trail of pheromones that the other ants can follow to reach the food source too. The shorter the path, the more pheromones they deposit. Ants are stochastic agents that construct a solution based on information they receive and the pheromone trail they perceive. After a while, good solution components, typically the edges on a graph, have more pheromones and the ants are biased to use them. ACO has been used to tackle a plethora of problems in operations research, and in many other fields, notably the well-known Travelling Salesman Problem (TSP).

The importance of efficient tool management in automated manufacturing systems has been extensively stressed (Gray, Seidmann, & Stecke, 1988). Tooling and sequencing problems were first identified in the metal working industry where a numerically controlled forging machine with two eight-slot magazines was in use. The tools were heavy and took a significant time to be switched (Blażewicz, Finke, Haupt, & Schmidt, 1988). Another application is in Printed Circuit Board manufacturing where a machine can be equipped with different electronic component feeders that can be regarded as tools (Bard, 1988). The problem is also present in other fields than operations, for instance in memory management in computer systems (McGeoch & Sleator, 1994). A recent case study was published about a South African printing company which owned a colour printer that was able to load up to eight different colours. It took them 30 minutes to wash a single cartridge when changing the colours (Burger, Jacobs, van Vuuren, & Visagie, 2015). The company used to schedule the print jobs on a first-in-first-out basis, leading to an average

of 103 washes to complete a twenty-job sequence. After optimizing the sequencing, the number of washes was reduced to 64, which resulted in a considerably shorter makespan.

This thesis will be organised as follows. In chapter 2 we will state the formal problem and its various properties. In the following chapters we will review the existing solving methods for the SSP found in the literature. Specifically, in chapter 3 we will review the exact approaches that aim to find an optimal solution; chapter 4 will focus on the heuristic approaches; chapter 5 will look at the metaheuristic approaches. In chapter 6, we will do an in-depth presentation of the Ant Colony Optimization class of metaheuristics, its various algorithms and their application to the TSP. Finally, in chapter 7 we will develop an Ant Colony algorithm for the SSP, tune it, and show that this is a strong approach to solve the problem.

Chapter 2

The Job Sequencing and Tool Switching Problem

In this chapter we will formally state the Job Sequencing and Tool Switching Problem. To effectively solve a problem, one must know its inner properties. To that end, we will establish the notation that we will use through this thesis in section 2.1; in section 2.2 we talk about the complexity of the problem and show how it can be decomposed; in section 2.3 we talk about the symmetry of the problem; in section 2.4 we present its lower bounds. We will also review the main variants of the problem in section 2.5.

2.1 Statement of the problem

Tang and Denardo (1988a), concurrently with Bard (1988), were the first to formalise this problem and the general notation that will be used throughout this thesis. Let $J = \{1, \ldots, N\}$ be the set of N jobs that must be sequenced and $T = \{1, \ldots, M\}$ be the set of M tools that are necessary to process the N jobs. We denote T_j as the set of tools needed to process the job $j \in J$, J_t is the set of jobs for which the tool $t \in T$ is needed and J_t^C is the complement of J_t, that is, the set of jobs that do not need the tool t to be processed. C is the capacity of the tool magazine, that is, the maximum number of tools that can be loaded simultaneously. A sequence σ is a permutation of J. It contains each job once, in a certain order, and therefore it is a feasible solution. The input parameters of a problem (a.k.a. an instance) are a binary matrix A of dimensions $(M \times N)$, where $a_{ij} = 1$ if job j requires the tool i to be processed and the capacity (C). A complete summary of the notation can be found at the end of this chapter.

An example of an instance retrieved from Tang and Denardo (1988a) shows a simple case in figure 2 along with its respective A matrix where there are 10 different jobs to process, 9 different tools but only the capacity to handle 4 of them at a time. We see in

this example that some of the jobs do not require 4 tools to be processed. Fortunately, the decision of which tools to load in the magazine other than the required ones is solvable in polynomial time for a given sequence using a greedy procedure called *Keep Tools Needed Soonest*. It will be explained in detail in section 2.2.2. Before modelling the problem, Tang and Denardo (1988a) as well as Bard (1988) who is more exhaustive, enumerated certain assumptions of the SSP.

1. Batch sizes are small. Thus the tool switching time is significant.
2. A job does not necessarily require C tools to be processed.
3. The capacity is sufficient to process any job at once. $C \geq \max_j\{|T_j|\}$.
4. The position of the tools in the magazine is irrelevant.
5. Switching costs are tool and application independent. It is equal to 1 for all tools and all jobs.
6. Only one tool is changed at a time.
7. All tools take one slot in the tool magazine.
8. A job can only be processed in one go regardless of whether some of its required tools have been loaded earlier in the sequence.

Jobs	1	2	3	4	5	6	7	8	9	10
Tools	1	1	2	1	3	1	1	6	3	5
	4	3	6	5	5	2				7
	8	5	7	7	8	4				
	9		8	9						

Magazine Capacity = 4

$$A = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2: A simple problem instance with 10 jobs, 9 tools and a capacity of 4.

9. The sequence is not cyclic, that is, the number of tool switches between job at instant N and job at instant 1 is irrelevant.

The only purpose of assumption 1 is to make the problem relevant: with long batches of production, the switching time is not significant. The other assumptions are made to simplify the problem and to represent the mechanic of this manufacturing system (Bard, 1988). They can be lifted in order to create variants of the SSP. This will be discussed later in section 2.5. Assumption 9 was never clearly stated in any paper but we prefer to make it clear from the beginning.

2.2 Complexity of the SSP

Some problems are known to be inherently difficult to solve. The SSP is one of them, it belongs to the NP-Hard class of problems. Informally, a problem is NP-hard if it is at least as hard as another known NP-hard problem (Garey & Johnson, 2002). Tang and Denardo (1988a) claim that the SSP is NP-Hard because it can be seen as a Travelling Salesman Problem with variable edge lengths. Since "finding the shortest Hamiltonian path on a complete graph is equivalent to solving the Travelling Salesman Problem, and since the TSP is a well-known NP-hard problem we conclude that the Job Sequencing problem is also NP-hard" (Tang & Denardo, 1988a, p. 770).

The TSP consists in finding the shortest route which goes through all the nodes of a graph exactly once on a complete graph (see figure 3). In fact, the SSP is reduced to a TSP if all the jobs need exactly C tools, that is, if $|T_j| = C$ for all $j \in J$, in which case the number of tool switches between two jobs is fixed and becomes a distance. Crama, Kolen, and Oerlemans (1994) present more thorough and formal proof of the NP-Hardness of the problem if $M > C \geq 2$ (otherwise the problem is trivial).

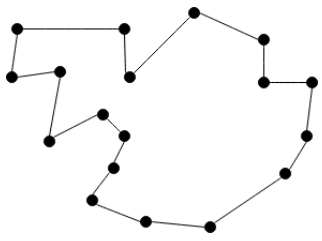


Figure 3: The Travelling Salesman Problem consists in finding the shortest path passing by every city exactly once.

2.2.1 Decomposition of the problem

The SSP can be decomposed in two nested subproblems. A problem is nested into another when it must be solved in order to compute the cost of a solution of the other. In other words, each solution of the high-level problem is a subproblem that must be solved.

The Job-Sequencing problem

The Job Sequencing subproblem is the high-level problem where an optimal job sequence must be found. This is the hard part of the SSP, "the hard nut to crack" to use the words of Crama et al. (1994). This high-level problem is the equivalent of a Travelling Salesman Problem: one must find an optimal order of jobs (instead of cities) yielding the minimum number of tool switches.

The Tool-switching subproblem (or tool replacement problem)

For a given job sequence, one must find the optimal tooling that gives its cost in terms of number of tool switches. This is the nested subproblem. Tang and Denardo (1988a) showed that the tool-switching subproblem can be solved to optimality in polynomial time by following a greedy policy called *Keep Tools Needed Soonest* (KTNS).

The objective of the SSP can therefore be rephrased as "find the sequence of jobs whose optimal tooling requires the minimum number of tool switches".

2.2.2 The KTNS procedure

For a given sequence of jobs, the *Keep Tools Needed Soonest* is an optimal policy to decide which unneeded tools to load in the free slots during the processing of a job. The name is self-explanatory: we keep the tools that were loaded at the previous instant and that will be needed the soonest in the rest of the job sequence. It can be easily shown that the time complexity of KTNS is $O(NM)$ (Tang & Denardo, 1988a).

Formally, it follows two rules:

1. When some tools must be removed, the selected useless tools to keep are the ones that will be needed the soonest in the rest of the job sequence.

2. Tools are loaded if and only if they are needed for the current job. No tool is ever loaded in prevision of future needs.

To prove that this procedure leads to an optimal solution, Crama et al. (1994) use the concept of 0-blocks minimization. Let P be any column permutation of the matrix A, it represents a sequence of jobs. Then, $P_{ij} = 1$ if tool i is required for the j^{th} job of the

sequence (at instant j), 0 otherwise. So P is a feasible solution where only the needed tools are loaded, which means that all the unneeded tools are unloaded. For an instance with 3 tools and a capacity of 2, the matrix could look like

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

The authors define a 0-block as a suite of several zeros in a row that are bounded by a 1 on each extreme. In this example, there are three 0-blocks, namely $\{(1,3),(1,4)\}, \{(3,2)\}$ and $(3,5)$. Formally, a 0-block is a set of the form $\{(i,j),(i,j+1)....,(i,j+k)\}$, for which the following conditions hold:

1. $1 < j \leq j + k < N$
2. $P_{ij} = P_{i,j+1} = ... = P_{i,j+k} = 0$
3. $P_{i,j-1} = P_{i,j+k+1} = 1$

Let $c_j = C - \sum_{i=1}^{M} P_{i,j}$ be the number of available slots for a job j, in other words, the unused capacity. In the example above, $c_3 = c_4 = c_5 = 1$. Intuitively, a 0-block is a maximal time interval before and after which tool i is needed, but during which it is not. It is easy to see that each 0-block in P is associated with an extra setup of a tool. Flipping 0-blocks to 1 when $c_j > 0$ for all jobs in the block reduces the number of these extra setups. Thus, we can rephrase the tooling problem as "flip to 1 as many 0-blocks of P as possible, while flipping at most c_j entries in column j ($j = 1, 2, ..., N$)"(Crama et al., 1994, p. 38-39). For instance, the 0-blocks $\{(1,3),(1,4)\}$ and $\{(3,5)\}$ of the example above can be flipped in order to get the Tooling Matrix W.

$$W = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

If one sorts all the 0-blocks in non-decreasing order (the relevant value for the sorting being the position of the first job of the block and the last job to break ties), then flipping the 0-blocks greedily from the first to the last, but only under the condition that there is enough capacity left, leads to an optimal solution. This conclusion is derived from a more general problem presented by Hoffman, Kolen, and Sakarovitch (1985). It is quite straightforward to see that this is precisely what the KTNS procedure does. This operation requires a pre-computation of the 0-blocks.

Note that it is possible that the magazine is not fully loaded during the first jobs if they do not require C tools. Although, the KTNS presented by Tang and Denardo (1988a)

has a step that pre-loads non-needed tools for the first job so the magazine is always full. The notation they use to present the procedure is quite sloppy and needlessly complex. Hence, we decided to propose our own version, but the outcome is the same.

Denote the integer $L(i, n)$ as the first instant at or after instant n at which tool i is needed, if tool i is never needed after instant $n-1$, then $L(i, n) = N$. P_n is the n^{th} column of the permutation matrix P (i.e. the n^{th} instant), and $P_{i,n}$ is equal to 1 if tool i is needed at instant n. The KTNS procedure goes as follows.

Step 1. Set $n = 1$. Flip $C - \sum_i P_{i,n}$ values of $P_{i,n}$ from 0 to 1 having minimal values of $L(i, n)$. Break ties arbitrarily. Set $n = 2$

Step 2. Stop if $n = N$ and return P (renamed W) that now contains the optimal tooling.

Step 3. If $\sum_i P_{i,n} = C$, set $n = n + 1$ and go to Step 2.

Step 4. Pick i such that $i = argmin_k\{L(k, n) : P_{k,n} = 0, P_{k,n-1} = 1\}$, set $P_{i,n} = 1$. Go to Step 3.

As explained, step 1 is a special case instant that fills the empty slots at instant 1 with the tools needed in the soonest future jobs. Steps 3 and 4 fill the magazine with tools loaded at the previous instant and needed the soonest.

Back to the NP-hardness of the entire problem, we understand that if all the jobs need C tools to be processed, there is no need to perform the KTNS. Indeed, there is no liberty regarding the tools to be loaded for a given sequence, no 0-block can be flipped. Hence, the cost of an edge will always be the same: the number of different tools between the nodes it links. In this case, the Tool Switching subproblem is eliminated, and the Job Sequencing subproblem is nothing more than a TSP. That way, the problem is reduced to a known NP-hard problem and is proven to be NP-hard as well.

2.2.3 The dominance rule

Tang and Denardo (1988a) also discussed the *dominance rule* which intervenes when one job has a set of required tools (T_j) that is a subset of another ($T_j \subseteq T_i$). It is intuitive to understand that in this case, it is always optimal to have those jobs following the "bigger" job, as no tool switch is needed. In fact, if those jobs are removed from the problem, its optimal solution remains the same. This property can therefore be used to reduce the complexity of an instance. For example, the instance in figure 2 can be reduced from 10 to 6 jobs. Indeed $T_{10} \subseteq T_4$ and jobs 7, 8 and 9 are subsets of several other jobs. The new instance is then as shown in figure 4. It is particularly important to take this into account when generating random problems, in order not to have dominated jobs.

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

Magazine Capacity = 4

Figure 4: A reduced instance.

2.3 The symmetry property

Ghiani, Grieco, and Guerriero (2007) emphasized a property of the SSP called the *symmetry property*. It states that for any sequence of jobs, the reverse of this sequence will always have the same minimal number of tool switches. This property can be used to reduce by half the number of sequences to compute. They notably use it in the same paper for a Branch & Bound algorithm and in their later work for a Branch & Cut algorithm (Ghiani, Grieco, & Guerriero, 2010).

Formally, if f_1 and f_2 are the minimum number of tool switches for the job sequences $(j_1, \ldots, j_N) \equiv s_1$ and $(j_1, \ldots, j_N)^R \equiv s_2$ respectively, where the superscript R means that the sequence is reversed such that j_N is the first job, J_{N-1} the second, etc. It results that $f_1 = f_2$.

2.3.1 Proof

Although the property could be verified by experimentation, the authors provide a formal proof of their theorem. Let W_n be the set of tools loaded at instant n in the sequence σ_1, this is the output of the KTNS algorithm. The mathematical expression of the optimal number of tool switches is

$$\sum_{n=2}^{N} |W_n \backslash W_{n-1}| = f_1 \tag{2.3.1}$$

Note that this expression ignores the tools mounted during the first job (W_1). To be consistent with the definition of Tang and Denardo (1988a) "$+|W_1|$" should be appended to the left-hand side of the equality, or a dummy job 0 should be added to allow the sum to start from n=1. But we will leave it as it is in the paper, because it does not matter for the proof because we assume that $|W_n| = C, \ \forall n$.

Likewise, let W_n^R be the set of tools loaded at instant n in the sequence σ_2.

$$\sum_{n=2}^{N} |W_n^R \backslash W_{n-1}^R| = f_2 \tag{2.3.2}$$

They proceed by contradiction. Let's assume that $f_1 < f_2$, reversing the tooling of σ_1 gives a feasible solution to $\sigma_2 := (W_1^R = W_N, W_2^R = W_{N-1}, \ldots, W_N^R = W_1)$. Because $|W_n| = C,\ \forall n$, we know that

$$|W_n \backslash W_{n-1}| = |W_{n-1} \backslash W_n| \tag{2.3.3}$$

So

$$\sum_{n=2}^{N} |W_{n-1} \backslash W_n| = \sum_{n=2}^{N} |W_n \backslash W_{n-1}| = f_1 \tag{2.3.4}$$

That means that we created a solution for s_2 that yields $f_1 < f_2$ switches. This would violate the optimality of the KTNS, which has been proved. With the same reasoning for the opposite case with $f_2 < f_1$, we conclude that the only possibility is that $f_1 = f_2$.

2.4 Lower bounds

Lower bounds are expressions that define the minimum cost that can be achieved by completing a partial or empty solution. They allow to identify partial solutions that are guaranteed not to be able to give a solution with a lesser objective than the current best solution found. If an algorithm finds a solution that has the same cost as the global lower bound, that is, a lower bound on the empty solution, then this solution is optimal, and the procedure can be stopped. Every mixed integer program is lower bounded by its linear relaxation. However, it can sometimes be of a very poor quality (e.g. the linear relaxation of the model in Tang and Denardo (1988a) yields a solution of 0). With all the integrality constraints relaxed, the optimal solution of the obtained linear program is a global lower bound on the problem. With the SSP, another easy global lower bound is M: by definition, every tool has to be used at least once and thus loaded at least once. However, achieving M tool switches in a sequence requires a very large magazine for most problems.

Laporte, Salazar-Gonzalez, and Semet (2004) provide two more complex lower bounds for incomplete sequences. These lower bounds can be computed on partial sequences during a construction or a tree search procedure. For instance, they use them in their Branch & Bound algorithm (see chapter 3). If the lower bound exceeds the current optimal solution, then the search algorithm can prune all the solutions that begin with the partial sequence.

Given a partial sequence of jobs $s = (j_1, \ldots, j_p)$ a lower bound on the optimal value is obtained by computing the minimum number of tool switches to process the jobs in the partial sequence (obtained with the KTNS) plus the maximum of the two following lower

bounds.

Lower bound 1. The number of tools required by the last job (j_p) plus the number of different tools required by the remaining jobs ($Q = J \backslash s$) minus the capacity.

$$|T_{j_p} \cup (\cup_{j \in Q} T_j)| - C \tag{2.4.1}$$

Computing this bound has a time complexity of $O(M)$.

Lower bound 2. The cost of the minimum j_p-spanning tree on the edges of $\{j_p\} \cup Q$. An j_p-spanning tree is a spanning tree on Q plus the least cost edge connecting it to j_p. The cost of an edge is $l_{ij} := \max\{0, |Ti \cup Tj| - C\}$. The matrix of the edge costs can be precomputed in $O(n^2)$ time. Then the lower bound is found in $O(n^2 \log(n))$ time if the Kruskal algorithm is used (Kruskal, 1956).

None of these lower bounds strictly dominates the other, which is why both are computed and the higher one is kept.

2.5 Versions

There exist many variants of the SSP. It is interesting to present some of them as many real applications will have particularities. However, since the complexity of the problem lies in the job-sequencing subproblem, the methodology for solving those problems will often be similar.

Tang and Denardo (1988b) introduced a variant in the second part of their work. This time, the objective is to minimize the number of *switching instants*. A switching instant is the time after having processed the nth job but before the tools are switched, that is, the moment at which the machine is stopped to change the tooling. This variant is relevant when all the tools can be switched at the same time instead of one by one. The job sequencing problem then becomes the *job grouping problem.* Here the jobs are grouped in classes. A class is a set of different jobs that can be processed with a total number of different tools that is smaller than C. A feasible solution for this problem is then a set of classes containing in total each job exactly once. This solution is optimal if the set contains the smallest number of classes possible. It is shown in the paper that this problem is also NP-hard, and that it is in fact a bin packing problem with overlapping more than a sequencing problem.

Crama, Moonen, Spieksma, and Talloen (2007) studied a case of the tool switching problem where the tools have a size expressed in number of slots. This is often the case in practice (Jain, Johnson, & Safai, 1996; Matzliach & Tzur, 2000; Stecke, 1983). The tools

must be loaded in the machine in slots that must be adjacent, so now the position of the tool in the magazine becomes important. In Crama et al. (1994), they demonstrate that the tool replacement subproblem is also NP-hard for non-uniform tool sizes. Those results are valid for round (i.e. slot 1 and slot C are adjacent) or straight magazines. Another variant introduces Loading and Unloading times (L/U) that depend on the tool that is removed and the tool that is inserted, more commonly named *changeover times*. Privault and Finke (1995) showed that the Tool Switching Problem remains solvable to optimality in polynomial time with such arbitrary changeover times.

The problem can also be generalized to multiple machines. The objective becomes to minimize the maximum number of tool switches of all the machines (Bard, 1988). Machines can be identical or have different magazine sizes. Finke and Kusiak (1987) investigated a case where several subsets of tools are acceptable to perform some jobs. Balakrishnan and Chakravarty (2001) developed an opportunistic resequencing to take advantage of unscheduled machine downtimes during which tool switches are "free" and a rescheduling can be performed for the remaining jobs.

2.6 Summary

We introduced the notation that will be used in this thesis. A summary can be found in figure 5. We presented the various properties of the SSP:

1. It consists in a Job Sequencing subproblem similar to a Travelling Salesman Problem.

2. Each sequence is a Tool Switching subproblem that can be solved in polynomial time using a very important procedure called the *KTNS*.

3. The SSP is a symmetrical problem.

We also reviewed the different lower bounds on the objective of the global problem and of partial sequences. Finally, we listed the main variants of the problem.

- $J = \{1, \ldots, N\}$ is the set of N jobs to be processed.
- $T = \{1, \ldots, M\}$ is the set of M tools needed for processing the jobs.
- $J_t \subseteq J$ is the set of jobs that require tool t to be processed.
- $J_t^C \subseteq J$ is the set of jobs that do not require tool t to be processed.
- T_j is the set of tools that job j requires to be processed.
- $\sigma \subseteq J$ is a partial or complete sequence/solution.
- C is the capacity of the magazine of the machine.
- A is the binary matrix of dimensions $M \times N$ where $a_{ij} = 1$ if tool i is needed to process job j.
- P is a column permutation of A. It is a solution to the problem.
- W is what we call P after it has been optimized using the KTNS procedure.
- W_i is the set of tools loaded at instant i after P has been optimized using the KTNS procedure.

Figure 5: A summary of the notation used in this thesis.

Chapter 3

Exact solution approaches

The objective of the SSP is to find the sequence and tooling that requires the minimum number of tool switches. Since the problem is NP-Hard, there exists no algorithm that gives a guaranteed optimal sequence in polynomial time unless $P = NP$. The simplest approach is the complete enumeration: try every single possible sequence, and keep the best one. With $N!$ possible sequences that must be solved with the KTNS procedure, this approach becomes quickly terrible as the number of possible sequences grows exponentially with the number of jobs. To overcome this, more intelligent algorithms exist to solve discrete optimization problems. The *Branch & Bound* algorithm is one example (Lawler & Wood, 1966). Branch & Bound schemes start from the root node of a tree by solving a linear relaxation of the problem, that is, all the variables are allowed to be fractional instead of integers. Unlike mixed integer problems, linear problems are solvable in polynomial time (Nesterov & Nemirovskii, 1994). By definition the optimal value of a relaxation is always a lower bound on the complete problem. The branching consists in fixing one variable to different integer values: a child node is created for each value. These new problems are therefore slightly more constrained. If a problem has a solution worse than the incumbent best feasible solution found until now, an upper bound on the optimal solution for a minimization problem, then fixing any other variable will only result in worse solutions and the entire subbranches can be pruned: they are not generated. The algorithm stops when the difference between the incumbent best solution and the lower bound on the optimal solution reaches zero, that is, when the entire tree is explored or pruned. The *Branch & Cut* algorithm uses the same scheme but instead of branching it can take the decision to add some valid constraints (cutting planes or constraints that were relaxed) that are violated by the solution at the node.

With a mixed integer programming formulation, it is always possible to perform a Branch & Bound procedure and try to find the optimal solution, if one exists. In this chapter we will present three formulations proposed by different authors. We will begin

with the original and most intuitive formulation, then a formulation based on the resemblance of the SSP with the TSP. The third one is an improved formulation designed to give higher linear relaxations in a Branch & Bound scheme. Finally, we will mention the other exact approaches in the last section.

3.1 Basic formulation

The first formulation was given by Tang and Denardo (1988a). Let us denote $K = \{1, 2, \ldots, N\}$ as the set of instants in the job sequence to be determined. In other words, an instant is a column in the permutation matrix P (see section 2.2.2), the instant of a job is thus its position in the sequence. We also denote:

- u_j^k as a binary decision variable equal to 1 if job j is processed in k^{th} position.
- v_t^k as a binary decision variable equal to 1 if tool t is loaded in k^{th} position.
- w_t^k as a binary decision variable equal to 1 if tool t is loaded in position k but was not in position $k-1$.

Formulation 1 - (Tang and Denardo, 1988a)

$$\min \sum_{t \in T} v_t^1 + \sum_{t \in T} \sum_{k \in K \setminus \{1\}} w_t^k \tag{3.1a}$$

$$s.t. \sum_{j \in J} u_j^k = 1 \qquad \forall k \in K \tag{3.1b}$$

$$\sum_{k \in K} u_j^k = 1 \qquad \forall j \in J \tag{3.1c}$$

$$\sum_{j \in J_t} u_j^k \leq v_t^k \qquad \forall k \in K, t \in T \tag{3.1d}$$

$$\sum_{t \in T} v_t^k \leq C \qquad \forall k \in K \tag{3.1e}$$

$$v_t^k - v_t^{k-1} \leq w_t^k \qquad \forall k \in K \setminus \{1\}, t \in T \tag{3.1f}$$

$$u_j^k \in 0, 1 \qquad \forall k \in K, j \in J \tag{3.1g}$$

$$v_t^k, w_t^k \in 0, 1 \qquad \forall k \in K, t \in T \tag{3.1h}$$

The objective function (3.1a) minimizes the tool switches between jobs plus the loading of the magazine at instant 1. Constraints (3.1b) and (3.1c) ensure that each job is processed exactly once. Constraints (3.1d) ensure that the required tools are loaded when a job is assigned to a position in the sequence. Constraints (3.1e) enforce the capacity of

the magazine. Constraints (3.1f) impose to count a tool switch if a tool is loaded between two instants. The remaining constraints define the domains of the variables.

This first model provides rather disappointing results when implemented in a Branch & Bound algorithm. Laporte et al. (2004) explain that this is because the linear relaxation of this formulation always has a solution equal to zero (i.e. zero switches needed). The solution $u_j^k = \frac{1}{N}$ for all j, k; $v_t^k = \frac{|Jt|}{N}$ for all k, t and $w_t^k = 0$ for all k, t is optimal for the linear relaxation. As a consequence, solving the SSP using this model with a Branch & Bound requires almost complete enumeration because the lower bounds generated are not tight and rarely allow to prune some branches.

3.2 Hamiltonian tour formulation

Laporte et al. (2004) then propose an alternative formulation that yields a better linear relaxation. This formulation is based on the resemblance of the SSP with the TSP. It exploits the fact that the job sequencing component of the SSP corresponds to a tour if a dummy job zero that requires no tool, $T_0 = \emptyset$, is added. In this formulation we denote

- $J_0 = J \cup \{0\}$
- $J^i = J \backslash \{i\}$, $i \in J$
- $J_0^i = J_0 \backslash \{i\}$, $i \in J$
- G the complete graph having J_0 as vertexset
- $x_{ij} = 1$ if job j follows job i in the sequence, equals 0 otherwise
- $y_j^t = 1$ if tool t is loaded on the magazine when job j is processed, equals 0 otherwise
- $z_j^t = 1$ if tool t is inserted to process job j

Constraints (3.2b) and (3.2c) are equivalent to constraints (3.1b) and (3.1c). Constraints (3.2d) prevent subtours. Constraints (3.2e) enforce the capacity of the magazine. Constraints (3.2f) enforce that a tool must be inserted or already loaded if it is needed for a job and (3.2g) is a special case with the dummy job 0. Constraints (3.2h) and (3.2i) ensure that a tool switch is only performed when the tool is required to execute a job immediately.

The authors propose two algorithms along with their formulation, a Branch & Bound and a Branch & Cut. Both use a broad branching rule instead of a classic binary tree: a node generates $|Q|$ branches, one for each possible subsequent job. Q is the set of jobs that have not been fixed at the parent node.

Formulation 2 - (Laporte et al., 2004)

$$\min \sum_{j \in J} \sum_{t \in T_j} z_j^t \tag{3.2a}$$

$$s.t. \sum_{j \in J_0^i} x_{ij} = 1 \qquad \forall i \in J_0 \tag{3.2b}$$

$$\sum_{j \in J_0^i} x_{ij} = 1 \qquad \forall j \in J_0 \tag{3.2c}$$

$$\sum_{\substack{i,j \in S: \\ i \neq j}} x_{ij} \leq |S| - 1 \qquad \forall S \in J_0, 2 \leq |S| \leq n-1 \tag{3.2d}$$

$$\sum_{t \in T} y_j^t \leq C \qquad \forall j \in J \tag{3.2e}$$

$$x_{ij} + y_j^t - y_i^t \leq z_j^t + 1 \qquad \forall i, j \in J, i \neq j, t \in T \tag{3.2f}$$

$$x_{0j} + y_j^t \leq z_j^t + 1 \qquad \forall j \in J, t \in T \tag{3.2g}$$

$$y_j^t = 1 \qquad \forall j \in J, t \in T_j \tag{3.2h}$$

$$z_j^t = 0 \qquad \forall j \in J, t \in T \backslash T_j \tag{3.2i}$$

$$y_j^t, z_j^t \in 0, 1 \qquad \forall j \in J, t \in T \tag{3.2j}$$

$$x_{ij} \in 0, 1 \qquad \forall i, j \in J_0 \tag{3.2k}$$

The Branch & Cut algorithm is based on a technique from Grötschel and Padberg (1985) for the TSP. It inserts the most violated subtour elimination constraints (3.2d) using a maximum-flow algorithm. An initial sequence is generated using a strong heuristic called GENIUS (Gendreau, Hertz, & Laporte, 1992; Hertz, Laporte, Mittaz, & Stecke, 1998). The authors only provide computational results for instances with 8 and 9 jobs and up to 25 tools.

For their Branch & Bound scheme, they make use of an initial upper bound generated by a simpler greedy heuristic. Lower bounding is not done using linear relaxations but with the two lower bounds presented in section 2.4. The branching rule consists in fixing the jobs in the order in which they appear in the greedy initial solution. The B&B was able to solve most of the instances with 15 jobs. They note that this formulation works better on instances that are close to a TSP, that is, when the minimum number of tools for a job is close to the magazine capacity. Ghiani et al. (2007) used the same B&B technique and formulation but took the symmetry property into consideration for the branching rule. They were able to solve instances with less than 15 Jobs, but still unable to solve larger instances. Adding the symmetry property only allowed a speed improvement.

3.3 Improved formulations

Catanzaro, Gouveia, and Labbé (2015) developed three improved integer programming formulations. They are proven to be tighter than the formulations we presented in the previous sections. Their formulations yield smaller gaps and shorter solution times than formulation 2 (Laporte et al., 2004). Formulation 3 (p. 20) is experimentally the best linear formulation for the SSP to this day: it is the result of several transformations of Formulation 2 such as tightening, reducing, or addition of valid inequalities. However, it could not solve larger instances. We will not explain the constraints in this thesis because they are the result of extensive mathematical operations. The interested reader can of course refer to the article (Catanzaro et al., 2015). In this formulation we denote $V_{ij}^t = J_t^c \cup \{i, j\}$ for all $i, j \in J, i \neq j$ such that $t \in T_i \cap T_j$.

3.4 Other formulations

Bard (1988) proposed a non-linear formulation. A later approach also based on a Branch-and-Cut algorithm was proposed by Ghiani et al. (2010) with a different non-linear formulation. They were able to solve some instances with up to 45 jobs and 30 tools to optimality. The authors believe that this success is mainly due to the introduction of the symmetry property which "allowed them to nearly halve the number of nodes in the search process, thus providing a remarkable speed up". However, these results should be taken with caution. Indeed, the random instance generation for $N \geq 25$ was setup such that $0.9 \times C \leq |T_j| \leq C$, making the problems fairly easy compared to other works where the 0.9 is set around 0.4 instead. These high minimum and maximum numbers of needed tools made the problems close to a TSP and thus the results may not be valid for the commonly used instance generation parameters proposed by Crama et al. (1994).

3.5 Summary

We reviewed the exact solution approaches for solving the SSP: a basic but intuitive formulation for Branch & Bound solving, a formulation based on the resemblance of the problem with the TSP, and an improved formulation tighter than the others. Some other non-linear methods were reviewed as well. None of the methods is able to solve problems with standard tool distribution with more than 15 jobs to guaranteed optimality.

Formulation 3 - (Catanzaro et al., 2015)

$$\min \sum_{j \in J} |T_j| x_{0j} + \sum_{\substack{i,j \in J: \\ i \neq j \\ t \in T_j \setminus T_i}} \left(x_{ij} - \sum_{\substack{k \in J: \\ k \neq j \\ t \in T_k}} f^t_{kj,ij} \right) \tag{3.3a}$$

$$s.t. \sum_{j \in J^i_0} x_{ij} = 1 \qquad \forall i \in J_0 \tag{3.3b}$$

$$\sum_{j \in J^j_0} x_{ij} = 1 \qquad \forall j \in J_0 \tag{3.3c}$$

$$\sum_{\substack{i,j \in S: \\ i \neq j}} x_{ij} \leq |S| - 1 \qquad \forall S \subset J_0,$$

$$2 \leq |S| \leq N - 1 \tag{3.3d}$$

$$\sum_{\substack{l \in V^t_{ij}: \\ l \neq k}} f^t_{ij,kl} - \sum_{\substack{l \in V^t_{ij}: \\ l \neq k}} f^t_{ij,lk} \geq 0 \qquad \forall i,j \in J : i \neq j,$$

$$\forall t \in T_i \cap T_j, \forall k \in J^c_t \tag{3.3e}$$

$$\sum_{i \in J_t} f^t_{il,kl} \leq x_{kl} \qquad \forall t \in T, \forall k \in J^c_t, l \in J_t \tag{3.3f}$$

$$\sum_{i \in J_t} f^t_{kj,kl} \leq x_{kl} \qquad \forall t \in T, \forall k \in J_t, l \in J^c_t \tag{3.3g}$$

$$\sum_{i \in J_t} f^t_{kj,kl} \leq x_{kl} \qquad \forall t \in T, \forall k, l \in J^c_t : k \neq l \tag{3.3h}$$

$$\sum_{\substack{t \in T: \\ t \in T_k \\ t \notin T_l}} \sum_{\substack{j \in J_t: \\ j \neq k}} f^t_{kj,kl} + \sum_{\substack{t \in T: \\ t \in T_k \\ t \notin T_l}} \sum_{\substack{i,j \in J_t: \\ i \neq j}} f^t_{ij,kl} \leq (C - |T_l|)\, x_{kl}$$

$$\forall k, l \in J : k \neq l \tag{3.3i}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in J_0 : i \neq j \tag{3.3j}$$

$$f^t_{ij,kl} \geq 0 \qquad \forall t \in T, \forall i,j \in J : i \neq j,$$
$$\forall k, l \in V^t_{ij} : k \neq l,$$
$$\{k,l\} \neq \{i,j\}. \tag{3.3k}$$

Chapter 4

Heuristic approaches

NP-hard problems quickly become impossible to solve to optimality even using Branch & Bound strategies because the size of the trees grows exponentially. Another approach is to use rules to generate solutions that are not guaranteed to be optimal, but believed to be good enough for real life applications. The so-called *heuristics* are typically faster than a tree search because they generate solutions in polynomial time. There are two main types of heuristic approaches. *Construction heuristics* build solutions (in our case job sequences) starting from nothing and follow a more or less simple behaviour, like being greedy. *Local searches* however try to find better solutions by modifying an existing one. A common strategy is to generate a solution with a greedy heuristic then use a local search to find the local optimum in its neighbourhood.

In this chapter, we will gather the diverse heuristic procedures developed for the SSP, author by author. For the sake of being complete, other approaches than construction and local searches will also be briefly reviewed.

Crama et al. (1994) provide by far the richest source of simple heuristics. They list six different categories. Four of them are construction strategies: Travelling Salesman heuristics, block minimization heuristics, greedy heuristics and interval heuristics. The two other categories are local searches. We will only present the most interesting of them in sections, 4.1 to 4.3. In section 4.4 we will present a more complex heuristic called GENIUS. Finally, in section 4.5 we briefly survey some other approaches.

4.1 Travelling Salesman Heuristics

They are based on an idea suggested by Tang and Denardo (1988a) to consider a graph $G = (V, E, lb)$ where V is the set of N jobs, E is the set of all pairs of jobs (edges) and $lb(i, j)$ is the length of the edge (i, j), defined as an underestimate, a lower bound, of the number of tool switches needed between the two jobs. Precisely:

$$lb(i,j) = max\{|T_i \cup T_j| - C, 0\}$$

The SSP is thus approximated to a symmetric Travelling Salesman instance. Hence, any heuristic for the TSP can be used on this graph to generate solutions that can be translated back to a SSP. The authors tried some well-known algorithms for the TSP on this graph:

1. *Shortest-edge heuristic.* The initial solution is found by always choosing the next job as the one with the lowest edge length. This greedy approach ignores all the future and past tooling decisions. This is the simplest non-random heuristic we can think of. It has a complexity of $O(N^2 \log N)$.

2. *Nearest neighbour, with every possible starting node.* See Golden and Stewart (1985). It has a complexity of $O(N^3)$.

3. *Farthest Insertion, with every possible starting node.* See Golden and Stewart (1985). It has a complexity of $O(N^4)$.

4. Any *Branch & Bound algorithm.* Solving the TSP instance to optimality (or with a small gap) will provide a solution to the SSP problem. However, this approach has an exponential time complexity as well since the TSP is an NP-hard problem. Finding an optimal solution to this TSP instance does not guarantee at all that the corresponding job sequence is optimal.

Among these, the Farthest Insertion is the procedure that has shown the best results for the SSP.

The *block minimization heuristics* work on the same principle except that they use the graph $G = (V, U, ub)$. V is the set of all jobs plus an additional dummy job (denoted by J_0) that requires no tool to be processed. This time the length $ub(i,j)$ is an upper bound on the number of tool switches between two jobs and is given by

$$ub(i,j) = |T_i \backslash T_j|$$

They are called block minimization because they are equivalent to trying to finding a TS path that minimizes the number of 1-blocks in the P matrix (c.f. section 2.2.2) (Crama et al., 1994).

4.2 Multiple-start Greedy Heuristic

This is a simple procedure that does not have the drawbacks of using lower or upper bounds. This procedure runs in $O(MN^3)$ time. It requires $O(N^2)$ applications of the KTNS ($O(MN)$):

Step 1. Start with a partial job sequence $\sigma = \{arbitraryStartingJob\}$; let $Q = J \backslash \sigma$.

Step 2. For each job j in Q, let $c(j)$ be the cost of the partial sequence $\sigma + \{j\}$, computed with the KTNS procedure.

Step 3. Set $\sigma = \sigma + \{i\}$ and $Q = Q \backslash \{i\}$ such that $i = argmin_j\{c(j) : j \in Q\}$.

Step 4. If Q is not empty, go to step 2. Else, σ is a complete solution.

Of course, this procedure should be executed N times, each time with a different starting job, resulting in a worst-case complexity of $O(MN^4)$. This was found to be the best construction heuristic among those proposed by Crama et al. (1994).

4.3 K-opt strategies

4.3.1 2-opt

2-opt is a famous heuristic for the TSP. Unlike the previous ones that were construction heuristics, this is a Local Search (LS) that tries to find better solutions in the neighbourhood of an initial complete one. The authors describe two versions of this approach, a global and a restricted one. The former, called the *global 2-opt,* was first proposed by Bard (1988) and is described in Crama et al. (1994) as:

Step 1. Find two jobs i and j whose exchange results in an improved sequence; if there are no such jobs, then return σ and stop; else, continue.

Step 2. Exchange i and j; call σ the resulting sequence; repeat Step 1.

This procedure has a complexity of $O(N^3M)$ because a KTNS must be performed $O(N^2)$ times to compute the cost of each new solution. Note that this local search is not actually a *2-opt* in the sense of Croes (1958). The definition of a *2-opt* move is deleting two edges and replacing them by two others. That operation is in fact the same as reversing a segment of the input sequence and simply computing its new cost. A job swap is a double *2-opt* move, that is, a *4-opt* move if the jobs are not subsequent. The difference is illustrated in figure 6. Hence these two local searches explore a different neighbourhood. In this thesis, what we will refer to as "2-opt" is the proper subsequence reversing operation as presented in procedure 1 and figure 7.

Procedure 1 2-opt for the SSP.

Input: σ //the initial sequence of jobs
  repeat $\leftarrow$ true
  while repeat **do**
    repeat $\leftarrow$ false
    for $i = 1, i \leq N - 1$ **do**
      for $j = i + 1, j \leq N$ **do**
        $\sigma' \leftarrow$ reverse the subsequence of σ starting from instant i to instant j
        if $cost(\sigma') < cost(\sigma)$ **then**
          $\sigma \leftarrow \sigma'$
          repeat $\leftarrow$ true
        end if
      end for
    end for
  end while
  return σ

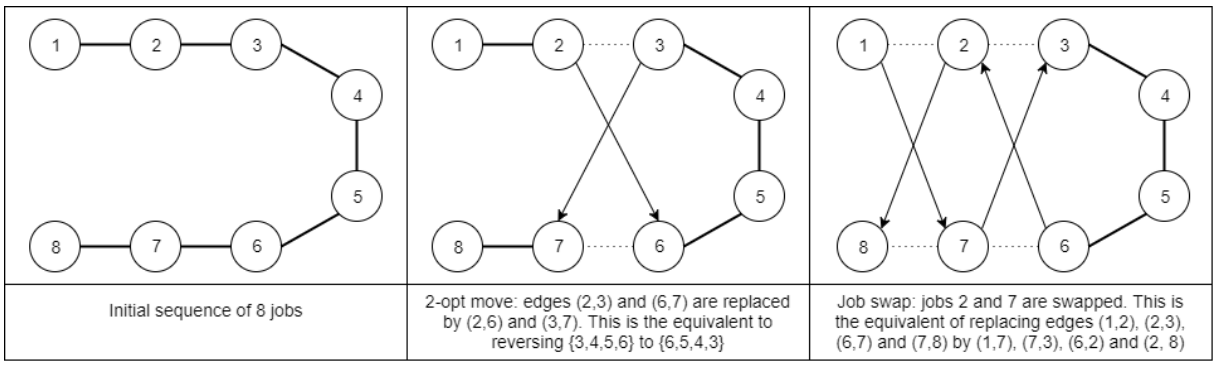


Figure 6: Illustration of the difference between a 2-opt move and a job swap.

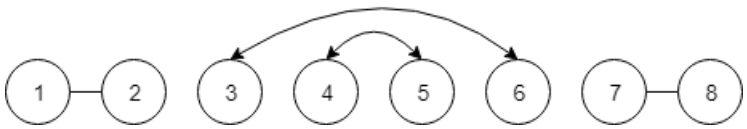


Figure 7: Reversing a subsequence necessitates to swap jobs by pairs.

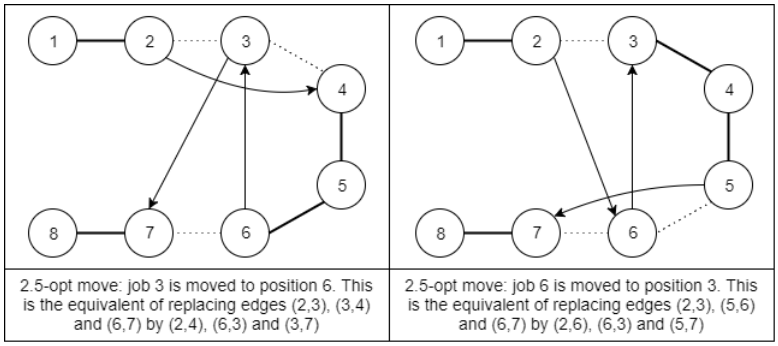


Figure 8: Two simple 3-opt moves.

4.3.2 2.5-opt plus job swap

2-opt can be extended to the *2.5-opt* by adding two simple 3-opt moves, that is, deleting three arcs and replacing them by three new ones. This is achieved by simply moving job i to position j or moving job j to position i (see figure 8). This allows to search in a wider space while keeping the $O(MN^3)$ complexity. Likewise, the *job swap 4-opt* move can be added as well by swapping jobs i and j and still keeping the same time complexity. This approach conciliates the *job swap* search with the *2.5-opt* and thus explores an even larger neighbourhood. We will refer to this as the *2.75-opt* in the rest of this thesis.

The 2-opt local search can basically be extended to any *k-opt*. The larger the neighbourhood, the more solutions are explored but at the expense of rapidly increasing complexity: $O(N^{k+1}M)$. The complete *3-opt*, for instance, requires 3 iterators over $\{1, \ldots, N\}$ instead of 2, that is, it multiplies by $O(N)$ the number of combinations of values that the iterators can take by adding a third *for loop*. Additionally, the number of possible moves increases exponentially with respect to k. The formula is $Moves(k) = 2^{k-1}(k-1)!$, including the moves that reinsert deleted arcs (Helsgaun, 2009).

4.4 GENIUS Heuristic

The GENIUS heuristic was imagined by Gendreau et al. (1992) for the Travelling Salesman Problem and adapted for the SSP by Hertz et al. (1998). It is a combination of Generalised Insertion (GENI), a constructive procedure, and Unstringing and Stringing (US), an improvement procedure.

The GENI algorithm can be summarized as follows. Starting from an arbitrary sequence of three jobs, insert an arbitrary remaining job among its p nearest neighbours. Distances between jobs can be approximated using the different metrics presented later in this section. The insertion is different from standard ones because it makes use of local

optimization similar to 3-opt or 4-opt among the neighbourhood. The method is efficient because it explores a limited number of possible insertions among the best candidates that are represented as the neighbourhood. The time complexity is $O(Np^4 + N^2)$. In practice p is small and should be between 3 and 7 (Hertz et al., 1998).

The US is a post-optimization procedure which takes off a vertex and inserts it back. The Stringing is exactly the same insertion as in GENI. The interest lies in the Unstringing which is essentially the exact opposite of Stringing, but it is done on a complete sequence. It can be applied to any tour regardless of the construction procedure used.

4.4.1 New metrics for TSP heuristics

Hertz et al. (1998) reviewed the heuristics presented in the previous sections and proposed a few improvements. Namely they addressed the Farthest Insertion (FI), the Nearest Neighbours (NN), the job swap neighbourhood search and the GENIUS heuristic.

These methods are applied on a TSP instance constructed by defining distances between the jobs. They propose a total of five different possible metrics. Three are simple lower bounds on the number of tool switches between jobs i and j:

$$d_1(i,j) = C - |T_i \cap T_j| \tag{4.1}$$

$$d_2(i,j) = |T_i \cup T_j| - |T_i \cap T_j| \tag{4.2}$$

$$d_3(i,j) = max\{|T_i \cup T_j| - C, 0\} \tag{4.3}$$

Note that d_3 is the metric used in Crama et al. (1994) for the TSP heuristics in section 4.1. Then they propose two distances that are improvements of d_3 and d_2. The metric d_4 aims to have a less myopic view of the problem by substracting more or less big quantities depending on the likeliness of the tools in T_i and T_j to be needed in the previous and next jobs. To do so, they compute $\lambda_k(i,j)$ as the number of other jobs than i and j that require tool k; and then $\Lambda(i,j) = \sum_{k \in T_i \cup T_j} \lambda_k(i,j)$. So d_4 substracts a quantity $[0, C]$ which is larger if the tools of i and j are frequent. Θ is a parameter in $[0, 1]$.

$$d_4 = max\left\{|T_i \cup T_j| - \left[\Theta \frac{\Lambda(i,j)}{(n-2)|T_i \cup T_j|}\right] C, 0\right\} \tag{4.4}$$

In the same spirit, they define d_5 as:

$$d_5(i,j) = \left(\frac{C+1}{C}|T_i \cup T_j| - |T_i \cap T_j|\right)\left[\frac{(n-2)|T_i \cup T_j|}{max\{\Lambda(i,j), 0.5\}}\right] \tag{4.5}$$

Additionally, they study the impact of using a Tool Switches objective instead of only using TSP metrics in these algorithms. This is the same idea that was used in *multiple-*

start greedy (see section 4.2). The distance metrics are now used in Farthest Insertion only to select the job to insert, and the KTNS is used to define exactly where in the partial sequence it should be inserted; as for GENIUS, the metrics are still used to compute the neighbourhood.

They compare the performances of these different metrics coupled with the different heuristics mentioned above, using or not the Tool Switches objective. They find that using the TS objective is slower but yields way better solutions. The metric d_5 is often the best and d_3 by far the worst. For fast solutions, they advise using the combinations FI/d_2, d_4 or d_5 or GENI/d_1, d_2, d_4 or d_5 without using the computationally intensive TS objective. If quality is the prime focus, then GENIUS/d_1 or d_2 coupled with the TS objective improvement yields higher quality solutions.

4.5 Other heuristics

Bard (1988) suggested a dual-based Lagrangian relaxation heuristic. By relaxing certain constraints of his non-linear formulation, he separates the job sequencing and the tool loading subproblems from each other without significantly altering the overall structure. The subproblems can be solved in $O(N^3)$ and $O(NM)$ time. This heuristic generates better solutions than using a Branch & Bound with the non-linear program proposed in the same article.

Privault and Finke (1995) addressed the SSP as a k-server problem solvable by the on-line *partioning algorithm*. In Privault and Finke (2000) they propose an alternative to the *greedy construction* proposed in Crama et al. (1994). They use an on-line tooling procedure called BSPA instead of using the KNTS at every construction step. Then the KTNS is used to obtain the real cost. This procedure shows better results than the Farthest Insertion. They also propose a grouping strategy that merges jobs that require few tools into big ones. That way, the lower bounds used in TSP heuristics are better approximations of the number of tool switches between two jobs.

Djellab, Djellab, and Gourgand (2000) introduced a heuristic called *iterative best insertion* based on a hypergraph representation of the SSP. It provides better solutions than the heuristics of Crama et al. (1994) but at the expense of computation time.

Zhou, Xi, and Cao (2005) used a classic beam-search to solve the problem. A beam-search is a restricted breadth search that prunes nodes on the search tree to only keep a certain number of the most promising ones at each level of the tree. They compare their results to those of Bard (1988) and conclude that their algorithm is more performant.

4.6 Summary

We reviewed the simple heuristic methods for solving the SSP. The two possible approaches are to approximate the problem as a Travelling Salesman instance or to use the KTNS instead of fixed distances. The latter approach is more complex but returns better solutions. We proposed a new *2.75-opt* for the SSP because the existing local searches were either weak or too complex.

Chapter 5

Metaheuristic approaches

Metaheuristics are solution methods that combine local improvement procedures and higher level strategies to create a process capable of escaping from local optima and performing a more thorough search of a solution space (Gendreau & Potvin, 2010). Unlike classic heuristics, they often do not have a deterministic outcome because they incorporate a stochastic component. The most classic examples (non-exhaustive) of metaheuristics are: Simulated Annealings, Tabu Searches, Genetic Algorithms, Greedy Randomized Adaptive Search Procedures, Ant Colony Optimization, Iterated Local Search, Large Neighbourhood Search, Guided Local Search, etc.

In this chapter we will review the most important metaheuristics that have been used to solve the SSP. Because metaheuristics find better solutions than the exact approaches in chapter 3 and the heuristics in chapter 4, the results presented in the papers will be used to benchmark our Ant Colony algorithm. We will devote the next chapter entirely to present Ant Colony Optimization for the TSP.

5.1 Tabu search

Al-Fawzan and Al-Sultan (2002) proposed a tabu search for the SSP as well as several modifications to it. This procedure was introduced by Glover (1986). It starts from an initial solution and then moves successively among neighbouring points. A candidate list is generated with a strategy that retains higher quality neighbours. The basic principle is to make a local search when a local optimum is encountered, allowing non-improving moves. The method avoids going back to previously visited solutions by forbidding (making tabu) the last moves by putting them in a *tabu list*. For instance, if the local search swaps Job i and Job j to create a worse solution, the local search will naturally want to swap them again, reversing the effect and thus cycling back. To avoid that, the swap (i, j) is made tabu. A simpler approach is to record previous solutions instead of moves. The size of

the tabu list can be either fixed or variable. Tabu search is not a heuristic procedure in essence, it is a strategy that can be combined with a local search to help it explore unvisited regions of the search space.

The primitive tabu search they propose only has a short-term memory component. The idea is simple:

Step 1. Generate an initial solution S.

Step 2. Generate k neighbours of S and identify the best one as $S_{\min}$.

Step 3. If $S_{\min}$ is in the tabu list then go back to Step 2. Other wise set $S = S_{\min}$ and add $S_{\min}$ to the tabu list.

Step 4. Stop the procedure if the maximum number of iteration is attained and return the best solution found so far. Otherwise go to Step 2.

The authors consider two methods to generate the neighbours. By randomly swapping two jobs of the current sequence or by moving a random block of a random size to a random new position. Then, they implement a long-term memory scheme by penalizing solutions with pairs of subsequent jobs that are often in the tabu list and by penalizing solutions with pairs that have been frequently swapped. A guidance mechanism is proposed by using a "strategic oscillation" and a *probabilistic oscillation* between the two neighbour generation methods in the hope of finding a new region of the solution space. The results of their experiments show that the long-term memory has a significant positive impact on the performance of the algorithm. The tabu search has also been applied later to the switching instants variant of the SSP (Konak, Kulturel-Konak, & Azizoğlu, 2008; Tang & Denardo, 1988b).

5.2 Evolutionary Algorithms

5.2.1 Memetic algorithm

Amaya, Cotta, and Fernández-Leiva (2008) proposed to tackle the SSP with a memetic algorithm. The authors published several papers on the subject, each time adding some features (Amaya, Cotta, & Fernández-Leiva, 2011, 2012, 2013). Memetic algorithms belong to the class of evolutionary algorithms as a form of population-based hybrid Genetic Algorithms (GA). This is a widely studied class of metaheuristics that are inspired by the natural selection mechanism (survival of the fittest). It is a population-based evolutionary approach that has many variants but the idea can be summarized as follows. A solution is an individual whose solution components constitute their chromosome; each component

is thus an *allele*. A fitness function evaluates the probability that an individual survives and reproduces with another. The mating creates the next generation of individuals by crossover of the chromosomes of the parents. There is also a chance of having random mutations in the genome of newborns. The evolution mechanism is simulated and it produces increasingly fitter individuals while vastly exploring the solution space. Memetic algorithms are hybrid GA in the sense that individuals are sometimes allowed to perform an individual learning to improve their genes by performing local searches or other improvement methods. The authors do not provide any performance comparison with other existing papers.

5.2.2 Biased Random Key Genetic Algorithm

Chaves, Lorena, Senne, and Resende (2016) proposed to use the Clustering Search and Biased Random Key Genetic Algorithm, a recent hybrid metaheuristic. The Cluster Search (CS) is a method proposed by Oliveira, Chaves, and Lorena (2013) that identifies promising regions of the search space by creating clusters composed of the best solutions generated by any heuristic procedure. It has four components:

1. A Search Metaheuristic (SM). It is any metaheuristic that generates diversified solutions. The good solutions generated are sent to the next component.

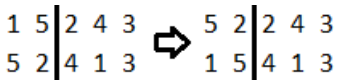
2. The Iterative Clustering (IC). A solution is associated with a cluster of other solutions according to a distance metric, here, the number of movements needed to transform the solution into the centre of the cluster. The centre of the cluster is updated when a solution is added to it.

3. The Analyzer Module (AM). This module detects when a cluster becomes promising, i.e. when the SM generated a certain quantity of solutions that were associated to this cluster.

4. The Local Search Module (LS). This module can be any kind of Local Search. It is applied to the centre of a cluster that has been detected promising to conduct a thorough search of the surrounding solution space.

The Biased Random Key Genetic Algorithm (BRKGA) constitutes the SM part of the Clustering Search. It is a variant of the Random Key Genetic Algorithm (Bean, 1994) class where one of the parents used is biased to be of higher fitness than the other (Gonçalves & Resende, 2011). The returned solution that will be clustered is simply the fittest offspring of each generation. In Random Key Genetic Algorithms, the alleles of the individuals are coded into random real numbers in [0,1]. The introduction of random keys guarantees

the construction of feasible offspring, particularly for sequencing problems. Instead of swapping jobs (the alleles) during the crossover, the keys are swapped and then their position in the sequence when sorted determines which job they correspond to. The example shows how a swap that should have generated infeasible solutions with a job being scheduled twice is avoided. The invalid crossover



Is keyed as

$$(0.33, 0.91, 0.46, 0.75, 0.51) \equiv \{1, 5, 2, 4, 3\}$$
$$(0.84, 0.32, 0.64, 0.04, 0.48) \equiv \{5, 2, 4, 1, 3\}$$

And gives the feasible children

$$(0.84, 0.32, 0.46, 0.75, 0.51) \equiv \{5, 1, 2, 4, 3\}$$
$$(0.33, 0.91, 0.64, 0.04, 0.48) \equiv \{2, 5, 4, 1, 3\}$$

They conclude that their computational results seem promising, although they are only compared to an Iterated Local Search from a paper in Portuguese by the same authors (Chaves, Senne, & Yanasse, 2012).

5.2.3 Dynamic Q-learning Genetic Algorithm

Ahmadi, Goldengorin, Süer, and Mosadegh (2018) used a Dynamic Q-learning technique to guide a Genetic Algorithm (DQGA) by defining optimal actions to take in given states of the procedure. Q-learning is a reinforcement learning technique that rewards a stochastic algorithm when it takes decisions that lead to a good result. In this case, the said actions are whether to apply the crossover before or after the mutations and the state depends on the effect an action would have. Each state-action pair is referenced in a table that keeps its score updated. Hence, the algorithm builds a rule-based intelligent behaviour. The GA includes other features such as removal of twin individuals at each generation.

The results are slightly better than with the Iterated Local Search of Paiva and Carvalho (2017) that they consider to be the state of the art way to solve the SSP (see section 5.3) for the *average best solution found* among ten runs over each instance. Their algorithm found the new best known solution of 3 out of 160 instances of Catanzaro et al. (2015) datasets. On the other hand, DQGA provided a worse *average solution quality* than ILS and is thus only partially the "state-of-the-art" method. In other words, over multiple solves of the same problem, DQGA is likely to find the best solution, but for a single run ILS has a better expected solution quality.

5.3 Iterated Local Search

Paiva and Carvalho (2017) proposed the Iterated Local Search, based the one developed by Chaves et al. (2012), but with components that are tailored to the SSP. An Iterated Local Search iteratively perturbates a local minimum solution and applies a local search to it in order to hopefully find a better one. The initial solution is generated by a two-steps heuristic that first generated a sequence of tools based on a breadth-first search of a tool graph. A tool graph is a graph $G = (V, E)$ where the vertices are the tools and the weight of an edge is the number of jobs that require the two tools they link. The initial sequence is then constructed using a greedy procedure that allows the jobs with the most frequent tool to be inserted in the solution first and the next ones are then gradually added in the authorized tool list. This method is only used for the initial solution generation.

The local search is also of their invention. It is inspired by the 0-block heuristic (Crama et al., 1994) and they named it the *1-block grouping heuristic*. It searches the rows of the binary P matrix for tools with two or more 1-blocks and attempts to move the jobs of a block one by one to the instants before and after the other block. They do this for each pair of subsequent 1-blocks, and for each tool.

The computational results presented conclude that their method outperforms the CS+ BRKGA (Chaves et al., 2016). The results have been benchmarked with the instances proposed by Catanzaro et al. (2015). They attribute their success to the use of tailored heuristics for the SSP.

5.4 Summary

We reviewed the metaheuristic strategies that were implemented to solve the SSP. The approaches we reviewed use standard metaheuristic strategies often coupled with a customized component for the SSP. Even if some authors tried to innovate with some "novel components" in their algorithms. A Dynamic Q-learning Genetic Algorithm and an Iterated Local Search are the current state-of-the-art methods for solving the problem and the results presented in their papers will serve as a benchmark for our future Ant Colony Optimization algorithm.

Chapter 6

Ant Colony Optimization algorithms

Ant Colony Optimization (ACO) is a class of metaheuristics in swarm intelligence methods that are inspired by the behaviour of real ant colonies when they are foraging for food. It was first developed by Dorigo (1992) for his doctoral thesis. The algorithm recreates the pheromone trail that ants dispose when they successfully located a food source. When the next ants are also looking, they are biased in their own search to take paths where the concentration of pheromones is high. This pheromone trail constitutes a global memory for the algorithm and helps it to converge towards good solutions while allowing the ants to explore the solution space.

Ant Colony Optimization algorithms have a couple of particularities that make them unique among the field of metaheuristic procedures. The first is the pheromone trail left on the graph. It constitutes a long-term memory to guide the next ants in their search of solutions. When an ant finds food, it has to come back to the nest. While doing so it leaves pheromones along the path. The quantity they deposit is inversely proportional to its length. An ant exploring a graph of nodes and arcs must choose which node from its neighbourhood it will explore next. To make that decision, the ant has two types of information: the heuristic information that represents the attractiveness of each node and the concentration of pheromones on possible steps that represents the accumulated experience of the colony. On their own, ants are not smart, but their ability to interact with each other through pheromone deposit makes the colony a very complex and self-organised intelligent system. This stochastic behaviour extends the set of solutions they can explore.

The second feature is that ACO is one of the few methods that are constructive in their heuristic process (another example is GRASP). Indeed, ants are simple stochastic construction agents, at least in the classic algorithms. Each ant starts from scratch and builds an entirely new solution based solely on the information it can perceive. Most of the other metaheuristics are local search oriented. That is, they start from random or

seed solutions and each iteration will be the result of the state of the previous one.

Ant Colony Optimization belongs to the class of *model-based* approaches, whereas most of the other methods such as evolutionary computation algorithms, iterated local searches, simulated annealing, are *instance-based* approaches. *Instance-based* algorithms use solely the current situation to generate the next solution/population. A *model-based search* progressively creates a probabilistic model to generate solutions in regions that are likely to contain high quality solutions. ACO belongs to the latter class: together, the ants and the construction graph form the probabilistic model, and the pheromones constitute the adaptive parameters of the model (Dorigo & Stützle, 2004).

We emphasized several times that the SSP is closely related to the Travelling Salesman Problem. The TSP happens to be the very first application of ACO (Dorigo, 1992). In their book, Dorigo and Stützle (2004) extensively explain how the algorithm can be applied to this problem. In this chapter, we will introduce Ant Colony Optimization for the TSP by following the structure of the book. We will present Ant System, the basic algorithm, then its numerous extensions. Then we will discuss the possibilities of hybridization and the parallel implementation taxonomy. This will constitute a solid basis to then derive an implementation for the SSP in chapter 7.

6.1 Ant System for the TSP

As a reminder of the TSP: let $G = (V, E)$ be a graph where V is the set of n nodes and E the set of edges linking these nodes. The objective of the problem is to find the shortest route that visits each node exactly once. In this chapter, we will only consider a complete graph, that is, where the distance between two cities is finite.

Ant System is the simplest Ant Colony algorithm. It has inferior performances than its extensions but it is the base for their construction. Laying down its principles is essential to understand the concept of Ant Colony algorithms because they have the same general structure that goes as follows. At each iteration, a given number of ants is sent to build a set of solutions. When they are done, some local search can be performed to

Procedure 2 Ant System

```
Set parameters, initialize pheromone trails
while termination criterion not met do
    ConstructAntSolutions
    ApplyLocalSearch //optional
    UpdateTrails
end while
return Best solution found
```

improve the solutions. Then the pheromone update rules are applied. At the following iterations, the ants will take into account the pheromones to build their solutions based on the experience of the previous groups. A pseudo-code of the general structure of Ant System is shown in procedure 2.

Procedure 3 shows the details of the construction steps followed by an ant. At each step of the process, an ant stands on node i and must decide to which unvisited node it will go next. Because ants are stochastic constructors, they must calculate a probability of choosing each candidate node j of the neighbourhood for the next step. The probability that ant k chooses to go to node j from node i is derived using the *random proportional rule:*

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{l \in N_i^k} \tau_{il}^\alpha \eta_{il}^\beta} & if j \in N_i^k \\ 0 & otherwise \end{cases} \tag{6.1}$$

Where τ is the matrix of dimensions $(n \times n)$ that contains the amount of pheromone on each edge of the graph. It is initialised uniformly at a certain value τ_0 and updated at each iteration of the algorithm. τ_{ij} is the amount of pheromones present on the arc (i, j). η_{ij} is the heuristic information, a measure of the attractiveness of going from node i to node j. In the case of the TSP, it is often the inverse of the distance between i and j ($\eta_{ij} = \frac{1}{d_{ij}}$). α and β are parameters that give a relative importance to these two types of information. N_i^k is the neighbourhood of ant k when it is standing at node i, that is, the set of nodes that it has not visited yet.

Fitting parameters α and β is critically important. If α is set too high, the algorithm will quickly converge to a solution close to the initial random ones. The initial randomness of the exploration will have a strong effect on the solution found. On the other hand, a

Procedure 3 ConstructAntSolutions

Input: the graph $G = (V, E)$ and the matrix τ
  listOfSolutions $\leftarrow \emptyset$
  for all ant **do**
    firstCity $\leftarrow$ a random city in V
    Solution $\leftarrow$ {firstCity}
    $N \leftarrow V \backslash \{firstCity\}$
    while N is not empty **do**
      Compute probability of going to j for all j in N
      $k \leftarrow$ select next city using random proportional rule
      $N \leftarrow N \backslash \{k\}$
    end while
    Add Solution to listOfSolutions
  end for
  return listOfSolutions

too low value will turn the ants into random searchers where the ants were to only rely on the heuristic information to decide which node to go to next. They would progress randomly on the graph but biased by the heuristic information without any memory of the past.

At the end of their construction, each ant k returns C^k, the cost of the tour T^k it built. After the optional local search comes the pheromone update. This phase is where the different algorithms distinguish from one another. It consists in the evaporation of the pheromones on every edge and the deposit by the ants of $\Delta\tau_{ij}^k$ on the edges in their solution. Formally the rules for Ant System are:

$$\tau_{ij} = (1-\rho)\tau_{ij} \qquad \forall i,j \qquad (evaporation) \tag{6.2}$$

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{K} \Delta\tau_{ij}^k \qquad \forall i,j \qquad (deposit) \tag{6.3}$$

Where

$$\Delta\tau_{ij}^k = \begin{cases} \frac{1}{C^k} & if (i,j) \in T^k \\ 0 & otherwise \end{cases} \tag{6.4}$$

ρ is the evaporation factor, an arbitrary parameter and K is the number of ants generated at each iteration. The evaporation implements a form of *forgetting* of the bad decisions the ants took in the past. It is used to avoid a too quick convergence of the colony on a suboptimal region.

6.2 Extensions and variants of Ant System

Ant System is more of a conceptual algorithm. It provides quite unsatisfactory results (see section 6.3). But it illustrates the general idea of reinforcing apparently good arcs/edges of the graph. In this section we present the different strategies tried by researchers to improve the performances of Ant System.

6.2.1 Elitist Ant System

The idea of Elitist Ant System (EAS) is to provide a pheromone reinforcement to the best tour found by the colony so far, T^{bs} (Dorigo, 1992). The pheromone deposit rule (6.3) becomes:

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{K} \Delta\tau_{ij}^k + e\Delta\tau_{ij}^{bs} \qquad \forall i,j \qquad (EAS\ deposit) \tag{6.5}$$

where e is a parameter that defines the weight given to the best-solution-so-far and $\Delta\tau_{ij}^{bs} = 1/C^{bs}$ if $(i,j) \in T^{bs}$. The computational results in Dorigo (1992) show that this strategy allows AS to find better and faster solutions.

6.2.2 Rank-based Ant System

The Rank-based Ant System improvement was proposed by Bullnheimer, Hartl, and Strauss (1999) and is experimentally slightly better than EAS. The amount of pheromones that the ants are allowed to deposit decreases with their rank among the other ants in the same iteration. Each ant is sorted by increasing cost and is given a rank r corresponding to its position in the ranking. Only the $w-1$ best ants are allowed to deposit. The best-so-far solution now has a weight $e = w$ and the amount of pheromones deposited by the other ants is multiplied by a factor given by $max\{0,(w-r)\}$. Equation (6.3) is now:

$$\tau_{ij} = \tau_{ij} + \sum_{r=1}^{w-1}(w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs} \qquad \forall i,j \qquad (RBAS\ deposit) \qquad (6.6)$$

6.2.3 Max-Min Ant System

The Max-Min Ant System introduces four modifications to AS (Stützle, 1999; Stützle & Hoos, 1997, 2000):

1. Only the *iteration-best* ant is allowed to deposit pheromones, in addition to the best-so-far ant. $\tau_{ij} = \tau_{ij} + \Delta\tau_{ij}^{ib} + \Delta\tau_{ij}^{bs}$.
2. The amount of pheromones present on each arc is bounded by a maximum and a minimum value. $\tau_{\min} \leq \tau_{ij} \leq \tau_{\max}$
3. The pheromones are initialised at their maximum value. $\tau_0 = \tau_{\max}$
4. The pheromones are reinitialised when the algorithm encounters a stagnation or when no improvement has been found for a few iterations.

The first rule may quickly lead to a stagnation effect in which all the ants follow the same tour. Hence, the second rule will ensure that the best tour does not grow too strong and the unexplored edges do not end up with no pheromones because of the evaporation. The rule can also be to alternate between the iteration-best and the best-so-far deposit. Usually, the algorithm starts by only using the iteration-best to explore the solution space and gradually shift towards the best-so-far to focus on areas that yielded the best solutions. The third rule is combined with a low evaporation rate and has for objective

to favour the tour exploration at the beginning of the search. With the reinitialization of the fourth rule, the MMAS can also reset its best-so-far solution to avoid going again to the same solution. In their experiments, this is done when the average λ-branching factor becomes smaller than 2.00001 and if no improved tour has been found for more than 250 iterations.

The λ-branching factor is a measure of the distribution of the pheromones on the matrix that was introduced in Dorigo and Gambardella (1997a). If the concentration of pheromones is very low on almost all the arcs that surround a node i, then the freedom of choices for an ant at this node is small. The λ-branching factor of node i is defined as the number of incident arcs (i.e. all the arcs that are linking i to another node) that have a pheromone trail value $\tau_{ij} \geq \tau^i_{\min} + \lambda(\tau^i_{\max} - \tau^i_{\min})$. $\tau^i_{\max}$ and $\tau^i_{\min}$ are the maximum and minimum pheromone values found on the incident edges of node i. λ is a parameter in [0,1]. The average of all the branching factors of all the cities gives a measure of the search space. This value is in the interval [2, n-1] since a node needs at least one entering and one exiting arc to be used by the ants. A value of 2.00001 is thus very small.

6.2.4 Ant Colony System

Ant Colony System has three main differences with Ant Systems (Dorigo & Gambardella, 1997b, 1997a). First, it exploits more the search experience with a more aggressive construction rule called the *pseudorandom proportional rule*. It uses a probability q_0 to choose the best next node rule instead of using the *random proportional rule* (equation 6.1). Formally the ant k at node i goes to the city j such that:

$$j = \begin{cases} argmax_{l \in N^k_i}\{\tau_{il}\eta^\beta_{il}\} & if q \leq q_0 \\ J(\beta) & otherwise \end{cases} \tag{6.7}$$

where q is a random variable uniformly distributed in [0,1] and $J(\beta)$ is the outcome of the *random proportional rule* with $\alpha = 1$. ACS will generate solutions very close to the best-so-far solution, especially when q_0 is high.

The second difference is that the pheromone deposit and the evaporation are only performed on the best-so-far tour, making the time complexity of the pheromone update $O(n)$ instead of $O(n^2)$.

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho\Delta\tau^{bs}_{ij} \tag{6.8}$$

Instead of evaporating on every edge, the third rule makes the ants "consume" the pheromones while they construct their solutions. Which means that when they cross an arc (i, j), the pheromones on this arc are directly updated such that

$$\tau_{ij} = (1 - \xi)\tau_{ij} + \xi\tau_0 \tag{6.9}$$

Because of these rules, the algorithm never converges or stagnates. If all the ants follow the same path then its pheromones will converge to τ_0 because of the local evaporation rule. This ensures that such a behaviour is discouraged by strongly reducing the pheromones on these edges if they are exploited too much. The algorithm can also be speeded up by restricting the neighbourhood of an ant to its nearest neighbours. This improvement also generates better solutions (Gambardella & Dorigo, 1996). In his recent PhD Thesis, Gambardella (2015) explains how ACS coupled with local searches and improvements managed to match state-of-the-art algorithms for the TSP and the Sequential Ordering Problem (SOP).

6.2.5 Approximate Non-deterministic Tree Search

ANTS is an ACO algorithm that exploits ideas from mathematical programming. It was proposed by Maniezzo (1999). It exploits some ideas from Branch & Bound algorithms, hence the name "tree search". In fact, it can be extended to an exact solution algorithm, close to a Branch & Bound.

ANTS is the first Ant algorithm that uses lower bounds to compute the heuristic information. That way, the solutions that have a lower bound higher than the best-so-far solution can be discarded. The drawback is that computing those lower bounds can be computationally expensive. ANTS also uses a slightly different *random proportional rule*, which is faster to compute and which has only one parameter $0 \leq \zeta \leq 1$ instead of two (α and β in previous algorithms).

$$p_{ij}^k = \begin{cases} \frac{\zeta\tau_{ij}+(1-\zeta)\eta_{ij}}{\sum_{l\in N_i^k}\zeta\tau_{il}+(1-\zeta)\eta_{il}} & if\ j \in N_i^k \\ 0 & otherwise \end{cases} \tag{6.10}$$

Finally, ANTS does not incorporate an evaporation rule, but instead uses a new definition for $\Delta\tau_{ij}^k$ that increases the pheromones if the solution is better than a moving average of the former ones and decreases them otherwise. Formally:

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{K} \Delta\tau_{ij}^k \tag{6.11}$$

$$\Delta\tau_{ij}^k = \begin{cases} \theta(\frac{C^k - LB}{C^{avg} - LB}) & if\ (i,j) \in T^k \\ 0 & otherwise \end{cases} \tag{6.12}$$

θ is a parameter, LB is a global lower bound on the solution of the TSP, and C^{avg} is

the moving average of the p (a parameter) last solutions found by the algorithm.

6.2.6 Hyper-cube Framework

The hyper-cube framework for ACO was introduced by Blum, Roli, and Dorigo (2001). It automatically rescales the pheromone values to lie in the interval [0,1]. It is a hyper-cube framework because if a problem is represented by binary decision variables, then a solution is a corner of the n-dimensional hyper-cube. In the case of a TSP, it is a $|V|^2$-dimensional hyper-cube. In the ACO case, it is the pheromones that are bounded in $[0, 1]$. This framework showed a more robust behaviour of the ants on average. The equations (6.2), (6.3) and (6.4) from AS become

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho \sum_{k=1}^{K} \Delta\tau_{ij}^{k} \tag{6.13}$$

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{\frac{1}{C^k}}{\sum_{h=1}^{K}\left(\frac{1}{C^h}\right)} & if\ (i,j) \in T^k \\ 0 & otherwise \end{cases} \tag{6.14}$$

6.3 Performances of the different ACO algorithms

Dorigo and Stützle (2004) provide a computational benchmark of the solving performances of the different extensions, except for ANTS. The results show that AS performs way worse than its extensions. MMAS and ACS are the most competitive ones. ACS is very aggressive and finds way better solutions in a short time, while MMAS starts worse than AS but eventually provides the best solutions on the very long run. Figure 9 shows a graphical summary of their experiments.

The authors also provide a summary (figure 10) of the parameters they found to be the best for each algorithm. C^{nn} is the cost of a solution found to the problem by a nearest-neighbour algorithm, but it might as well be any other heuristic. In this table, m is the number of ants in the colony; we used the symbol K in this chapter to avoid confusion later with M being the number of tools in the SSP.

ACO has been applied to NP-hard problems of many kinds: Routing problems, Assignment problems, Scheduling problems, Subset Problems, Machine Learning problems and other miscellaneous problems such as the 2D-HP Protein Folding problem (Shmygelska, Aguirre-Hernandez, & Hoos, 2002). For an extended and detailed list of ACO applications to NP-hard problems, see Dorigo and Stützle (2004, pp. 153-211). Notably, Konak and Kulturel-Konak (2007) used a custom variant of MMAS for the Job Sequencing and Tool Switching Instants problem, the first variant of the SSP (Tang & Denardo, 1988b).

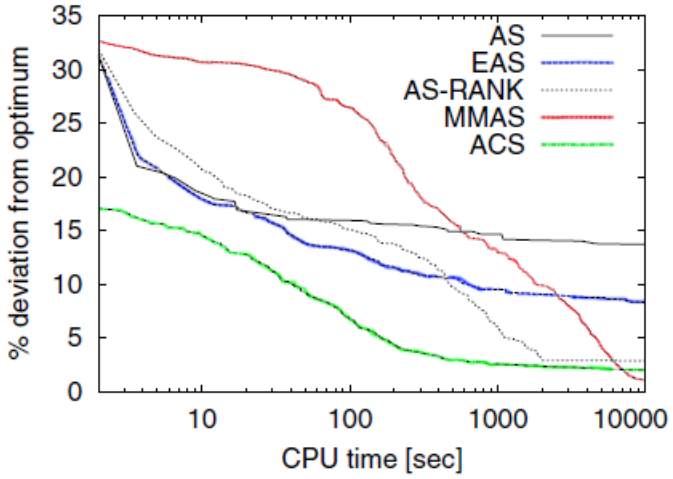


Figure 9: Comparison of AS and its extensions. The chart gives the average percentage deviation from the optimum for a TSP instance (rat783 from TSPLIB). The parameters are set as presented in figure 10 (Dorigo & Stützle, 2004, p. 95).

6.4 ACO with local searches

Local searches are particularly effective when they are coupled with a mechanism to generate initial solutions. ACO builds thousands of solutions in different neighbourhoods that are likely to be improved locally due to the stochastic nature of the construction phase. Dorigo and Stützle (2004) compare the three famous local searches *2-opt*, *2.5-opt* and *3-opt* (see section 4.3) for the TSP when used within an ACO algorithm. The results indicate that performing a local search on the solutions returned by the ants is an extremely effective strategy. As illustrated in figure 11, the larger the search, the better the solution provided in the long run, although with higher computational burden (3-opt is $O(n^3)$ and the others are $O(n^2)$).

The presence of local search can question the necessity of using heuristic information during the tour construction. Experiments show that MMAS performs slightly worse with $\beta = 0$ but ACS slightly better.

The last question they investigated is if we should deposit pheromones on the solution built by the ant or the locally optimized solution. Experiments prove that the Lamarckian approach (the latter) dominates by far the Darwinian (the former).

6.5 Hybridization with other strategies

It is also possible to hybridize an ACO algorithm with another metaheuristic. The implementation of a local search is already a hybridization in itself. Hadji, Rahoual, Talbi, and Bachelet (2000) used a Tabu Search to improve the solutions of the ants for the quadratic

Our experimental study of the various ACO algorithms for the TSP has identified parameter settings that result in good performance. For the parameters that are common to almost all the ACO algorithms, good settings (if no local search is applied) are given in the following table.

ACO algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	m/C^{nn}
EAS	1	2 to 5	0.5	n	$(e+m)/\rho C^{nn}$
AS_{rank}	1	2 to 5	0.1	n	$0.5r(r-1)/\rho C^{nn}$
$\mathcal{MM}$AS	1	2 to 5	0.02	n	$1/\rho C^{nn}$
ACS	—	2 to 5	0.1	10	$1/nC^{nn}$

Here, n is the number of cities in a TSP instance. All variants of AS also require some additional parameters. Good values for these parameters are:

EAS: The parameter e is set to $e = n$.

AS_{rank}: The number of ants that deposit pheromones is $w = 6$.

$\mathcal{MM}AS$: The pheromone trail limits are $\tau_{max} = 1/\rho C^{bs}$ and $\tau_{min} = \tau_{max}(1 - \sqrt[n]{0.05})/((avg - 1) \cdot \sqrt[n]{0.05})$, *where avg* is the average number of different choices available to an ant at each step while constructing a solution (for a justification of these values see Stützle & Hoos (2000). When applied to small TSP instances with up to 200 cities, good results are obtained by using always the iteration-best pheromone update rule, while on larger instances it becomes increasingly important to alternate between the iteration-best and the best-so-far pheromone update rules.

ACS: In the local pheromone trail update rule: $\xi = 0.1$. In the pseudorandom proportional action choice rule: $q_0 = 0.9$.

It should be clear that in individual instances, different settings may result in much better performance. However, these parameters were found to yield reasonable performance over a significant set of TSP instances.

Figure 10: Parameter Settings for ACO algorithms without local search (Dorigo & Stützle, p. 71).

assignment problem.

Another hybridization technique consists in making the ants start with a partial solution instead of empty ones, or to remove components of solutions found by previous ants: Wiesemann and Stützle (2006) proposed the iterated-ants algorithm that uses the Iterated Greedy algorithm (Ruiz & Stützle, 2007). Once some initial solution has been generated, IG iterates over construction heuristics by first removing solution components of a complete solution. From the partial solution a complete solution is then reconstructed using the standard solution construction of the underlying ACO algorithm. Computational results suggest that this idea is particularly useful if no effective local search is available.

Hybridization is also possible with Branch & Bound techniques. This has been done with ANTS (Maniezzo, 1999) and BeamACO (Blum, 2005, 2008) which combines a beam

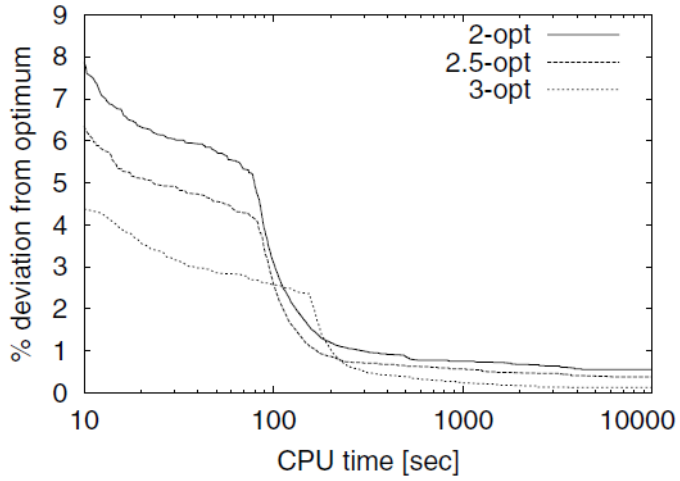


Figure 11: This plot gives the average percentage deviation from the optimal tour as a function of the CPU time in seconds for three different local searches. The algorithm is a MMAS solving a TSP instance "pr2392" from TSPLIB (Dorigo & Stützle, 2004, p. 95).

search with ACO. The latter algorithm is (or at least was) the state-of-the-art algorithm for the open shop scheduling problem. Finally, it is possible to combine ACO with constraint programming, but this is only interesting for highly constrained problems to which feasible solutions are hard to find (Meyer & Ernst, 2004).

6.6 Parallel implementations

Nowadays, even low tier CPUs contain multiple cores, allowing the execution of multiple threads simultaneously. Multiple-core CPUs can not only decouple the speed of a classic metaheuristic, but also introduces new exploration patterns to improve the results compared to independent sequential runs of a same procedure (Alba, 2005). Pedemonte, Nesmachnow, and Cancela (2011) surveyed the existing parallel implementation strategies for Ant Algorithms and proposed a new taxonomy to classify software-based parallel ACO algorithms. The taxonomy is composed of five main model categories.

- **The Master-Slave model** applies when a single master process manages the global information (i.e. pheromone matrix and best-solution-so-far) and controls a group of slave processes, in our case the ants. The model includes three subcategories for master-slave models depending on the amount of work the ants must do, namely *the granularity*. To be short, *coarse-grain models* create few parallel slave processes that each construct complete solutions. This is what the ACO algorithms does for the TSP. *Medium-grain models* create more ants that construct only some parts of

	# Colonies	
Cooperation	One	Many
No	Master-Slave	Parallel runs
Yes	Cellular	Multicolony

Figure 12: The different categories of parallel ACO algorithms.

the solution, which are then assembled by the master process. *Fine-grain models* are when the slave agents only process a single component of a solution.

- **The Cellular model** is a singular colony structured in small neighbourhoods each with an associated pheromone matrix. The neighbourhoods overlap as a means of communication between the substructures using a diffusion model.
- **The parallel independent runs model** simply runs multiple colonies in parallel, with identical or different parameters, and returns the best solution found among them.
- **The multicolony model** however implements a periodic information exchange between colonies, for instance sharing the pheromone matrices or the best-so-far solutions.
- **Hybrid models** implement ideas of more than one of the above-mentioned categories.

Although some computational results exist, there is no way to tell that one category is better than the others. The implementation of parallel ACO algorithms is extremely problem dependent. However, Pedemonte et al. (2011) observed that Multicolony models seem to be superior for most optimization problems.

6.7 Summary

There exist many Ant Colony algorithms for the TSP. We reviewed the common ones and explained in detail how they work. The most performing ones are the Min Max Ant System and the Ant Colony System. ACO algorithms can be hybridized with local searches, other metaheuristic strategies or with different approaches. Parallel implementations are promising with the recent advances of multi-core CPUs. We reviewed a taxonomy of the possibilities they offer.

Chapter 7

Ant Colony Optimization for the Job Sequencing and Tool Switching Problem

In this chapter we will implement an Ant Colony System to solve the SSP, test its performances and gradually modify it by introducing various mechanisms in the hope of improving the results. Among the different Ant algorithm variants presented, Min-Max Ant System and Ant Colony System are the ones that showed the most promising results for the TSP. Moreover, they are the only ones that have been tested with local searches. We will therefore begin by implementing these two and compare their respective performances.

7.1 Implementing MMAS and ACS for the SSP

The SSP is roughly a TSP with additional operations needed to compute the solution cost. Because of that, any Ant Colony procedure presented in chapter 6 can easily be adapted. The algorithms have quite a few parameters to choose. An obvious beginning point is to keep the same as those that were observed to be ideal for the TSP with a local search, as shown in figure 13. From there, we can try to find better suited ones to improve the results.

That said, some customizations are still needed for the SSP. First, the heuristic information must be redefined. In the SSP there exists no fixed distance between two jobs. In chapter 4, we reviewed many heuristics that use potential candidates for the heuristic information. The most obvious one is from the *multiple-start greedy construction* (Crama et al., 1994). The information used in this procedure is simply the cost of the current incomplete sequence plus an unvisited job. Formally, η_{ij} would be defined as

$$\eta_{ij} = 1/(c(j) - c(i)) \tag{7.1}$$

where $c(i)$ is the cost of the partial sequence of the ant and $c(j)$ is its cost if job j is added to it. Hence, just like one iteration of the *multiple-start greedy construction*, a single ant without local search has a complexity of $O(MN^3)$ because it performs $O(N^2)$ times the KTNS with $O(MN)$ complexity. Note that we cannot use $\eta_{ij} = 1/c(j)$ as in Crama et al. (1994) because as the sequence grows, the relative difference in cost between two possible jobs would be less significant. This would induce a non-homogenous behaviour of the ants during the construction because they would rely more on the pheromones when the sequence is almost complete than at the beginning of the construction.

The only ACO algorithms that have been applied with local search to the TSP are ACS and $\mathcal{MM}$AS. Good settings, obtained experimentally (see, e.g., Stützle & Hoos [2000] for $\mathcal{MM}$AS and Dorigo & Gambardella [1997b] for ACS), for the parameters common to both algorithms are indicated below.

ACO algorithm	α	β	ρ	m	τ_0
$\mathcal{MM}$AS	1	2	0.2	25	$1/\rho C^{nn}$
ACS	—	2	0.1	10	$1/nC^{nn}$

The remaining parameters are:

$\mathcal{MM}$AS: τ_{max} is set, as in box 3.1, to $\tau_{max} = 1/(\rho C^{bs})$, while $\tau_{min} = 1/(2n)$. For the pheromone deposit, the schedule for the frequency with which the best-so-far pheromone update is applied is

$$f_{bs} = \begin{cases} \infty & \text{if } i \leq 25 \\ 5 & \text{if } 26 \leq i \leq 75 \\ 3 & \text{if } 76 \leq i \leq 125 \\ 2 & \text{if } 126 \leq i \leq 250 \\ 1 & \text{otherwise} \end{cases} \tag{3.20}$$

where f_{bs} is the number of algorithm iterations between two updates performed by the best-so-far ant (in the other iterations it is the iteration-best ant that makes the update) and i is the iteration counter of the algorithm.

ACS: We have $\xi = 0.1$ and $q_0 = 0.98$.

Common to both algorithms is also that after each iteration all the tours constructed by the ants are improved by the local search. Additionally, in $\mathcal{MM}$AS occasional pheromone trail reinitializations are applied. This is done when the average λ-branching factor becomes smaller than 2.00001 and if for more than 250 iterations no improved tour has been found.

Note that on individual instances different settings may result in much better performance.

Figure 13: Parameter settings for ACO Algorithms with Local Search (Dorigo & Stützle, 2004, p. 96).

Unlike in the Euclidian TSP, there is a possibility that the cost between two jobs is 0. To avoid a division by zero, an arbitrary attractiveness value must be decided in that case. A very high value will bias the ants to systematically choose a node if the cost is zero; a lower value will only give them an advantage while still allowing the ant to explore different paths. The former solution restricts the ant to consider only the heuristic information and does not give any importance to the presence of pheromones. To avoid that, the arbitrary value we choose is $\eta_{ij} = 2$, which means that a job that does not require a switch is twice as attractive as if 1 switch was needed (for an equal value of pheromones).

Due to the nature of the SSP it is common to find various solutions with the same cost. The solution space of the problem is said to be "flat". There is no reason to only keep the first one found and not the others. For that reason, ACO algorithms should keep lists of the iteration-best and the best-so-far solutions instead of a single solution, and apply the pheromone update on all the solutions of the same value. That way, arcs that are in several best-so-far solutions accumulate more pheromones and the next generations of ants tend to build around these apparently really good arcs. This flatness of the solution space was already addressed the same way by Konak and Kulturel-Konak (2007) for the tool switching instant variant of the SSP (Tang & Denardo, 1988b). Considering multiple equivalent solutions is a legitimate concern: for instances with 40 jobs, the size of the best-so-far list can grow over 200 different best solutions. It would be a shame not to take them into account.

Ghiani et al. (2007) proved that a solution and its reverse have the same cost. So, in the same spirit, every time a new best-so-far solution is added to the list, we must add its reverse because we know that it has the very same cost. This basically allows to generate two ants for the price of one. Indeed if an ant generates a best-solution-so-far then its reverse can also be added to the best-solutions list. Note that the pheromone matrix will not necessarily be symmetrical nonetheless, for instance it will not be the case for ACS because of its local evaporation rule. A downside of these rules is the impossibility to use the λ-branching factor to assess if the algorithm converged because if there is more than one best-so-far solution the factor will never get close to 2.

To make an efficient use of multiple-core processors, the ants as well as the local optimization of their solutions were generated as threads and computed in parallel, reporting to the colony only when they generated a complete and locally optimized solution. Hence, our algorithms fall into the category of *Master-Slave coarse grain models*. Besides that, the considered ACO can be tried with the recommended parameters for the TSP (Dorigo & Stützle, 2004, p. 96). The local search used is the *2-opt* as presented in procedure 1 page 24.

		Subgroups			
	N, M	1	2	3	4
Groups A	10, 10	4	5	6	7
B	15, 20	6	8	10	12
C	30, 40	15	17	20	25
D	40, 60	20	22	25	30
		Capacity			

Table 1: This table shows the parameters of the problems in the different datasets. The groups (A, B, C and D) determine the N and M parameters, these are the rows. The subgroups (1, 2, 3 and 4) determine the capacity with respect to the group.

7.1.1 First computational results

The algorithms were coded in Java 9 and were run on a personal computer equipped with a dual-core (4 logic threads) i7-6500U processor and 8Gb of memory. The datasets used to benchmark the algorithms are provided in Catanzaro et al. (2015). They are used for benchmarking in the recent metaheuristic literature for the SSP. The instances were randomly generated according to the method in Crama et al. (1994). There are 16 datasets divided into four groups A, B, C, D that have (N,M) = (10,10), (15,20), (30,40) and (40,60) respectively. Each of these groups has four sets of 10 instances, where each set has a different capacity. The capacity increases for each dataset of a group, for example A1 has a capacity of 4, A2 of 5, etc, as shown in table 1. In total there are thus 160 problem instances. Instances in group A and most of group B are already solved to optimality, groups C and D only have a "best known value so far" that was computed in Catanzaro et al. (2015) using their exact solution approach, presented in section 3.3.

Groups A and B

Since instances from groups A and B are trivial and fast to compute, the stopping criterion will be a fixed number of 100 iterations for MMAS and 250 iterations for ACS or reaching a cost of M (a global lower bound on the number of tool switches). That way they generate the same number of ants with the parameters in figure 13. To produce similar results as in the Iterated Local Search (Paiva & Carvalho, 2017) and the Dynamic Q-learning Genetic Algorithm (Ahmadi et al., 2018), we solved each instance from groups A and B ten times. Then we computed the average of the solution found for each dataset (C), the average of the best of the 10 solutions found (C*) and the average time to execute the iterations (T). The results are reported in table 2.

MMAS achieved identical performances as ILS, whereas ACS failed to find solutions of the same quality every time for datasets B1 and B2. This might be explained by the

Dataset	MMAS			ACS			DQGA			ILS		
	C	C*	T	C	C*	T	C	C*	T	C	C*	T
datA1	**12.50**	12.50	0.42	**12.50**	12.50	0.51	**12.50**	12.50	0.17	**12.50**	12.50	0.09
datA2	**10.80**	10.80	0.31	**10.80**	10.80	0.38	**10.80**	10.80	0.17	**10.80**	10.80	0.09
datA3	**10.10**	10.10	0.05	**10.10**	10.10	0.07	**10.10**	10.10	0.19	**10.10**	10.10	0.05
datA4	**10.00**	10.00	0.01	**10.00**	10.00	0.00	**10.00**	10.00	0.22	**10.00**	10.00	0.04
datB1	**26.50**	26.50	1.28	26.58	26.50	1.38	26.72	26.50	1.62	**26.50**	26.50	1.09
datB2	**21.70**	21.70	1.46	21.74	21.70	1.77	21.74	21.70	2.12	**21.70**	21.70	1.38
datB3	**19.70**	19.70	0.77	**19.70**	19.70	1.02	19.76	19.70	1.39	**19.70**	19.70	1.12
datB4	**19.20**	19.20	0.02	**19.20**	19.20	0.01	**19.20**	19.20	0.79	**19.20**	19.20	0.71

Table 2: Comparison of the performances of MMAS, ACS, DQGA and ILS on datasets in groups A and B. Ant algorithms use a simple 2-opt. The results in columns ILS and DQGA are retrieved from Ahmadi et al. (2018, p. 20).

high $q_0 = 0.98$ coupled to a too simple local search (*2-opt*). This does not allow a wide search of the solution space unlike MMAS. Still, ACS performed better than DQGA in terms of average solution cost for datasets B1, B2 and B3. All the instances were solved to at least their best-known value found in Catanzaro et al. (2015). These problems are too easy to solve to be used for any kind of benchmark. The solutions found are probably solved to their optimum, hence we will not use them any further for benchmarking.

Groups C and D

Runs on non-trivial instances provided interesting results, reported in table 3. The first conclusion is that ACS seems to strongly outperform MMAS. If we take a deeper look at the ACS column, we can make two interesting observations. The first is that it found new best-known-solutions for some instances of D1, outperforming both DQGA and ILS with a lower C* for this dataset. On the other hand, ACS is still outperformed by ILS in terms of average-solution-quality (C) and by DQGA in terms of average-best-solution-quality(C*) for the other datasets.

The complexity of the SSP comes from the range of the impact of the previous choices of jobs on the following jobs to choose. The flexibility offered by the free slots of the magazine allows to strongly reduce the number of tool switches but also requires more intelligence to optimally make use of it. One explanation for these results is the myopic intelligence of the ants. The pheromones and the heuristic information only allow the ants to think about the very next step they will take. This behaviour is effective on instances close to a TSP (C1 and D1) because being myopic is fine but these first Ant Colony algorithms do not explore the less obvious permutations of jobs.

These conclusions are partly what drove us to attempt to find improvements for ACS to remedy that situation. Note that because we ran ACS and MMAS only two times

Dataset	MMAS			ACS			DQGA			ILS		
	C	C*	T	C	C*	T	C	C*	T	C	C*	T
datC1	100.45	100.10	900.39	**98.90**	**98.80**	900.14	99.82	**98.80**	145.29	99.09	98.90	99.47
datC2	84.65	84.50	900.52	**82.80**	82.60	900.18	83.74	**82.50**	196.09	82.82	**82.50**	137.08
datC3	69.00	68.70	900.57	67.20	66.90	900.19	67.82	**66.60**	220.32	**66.78**	**66.60**	172.59
datC4	53.05	52.80	900.80	51.70	51.60	900.34	52.10	**51.20**	144.89	**51.47**	51.30	137.79
datD1	203.45	203.20	901.38	**198.05**	**197.60**	900.52	200.16	198.00	213.43	198.36	198.00	488.66
datD2	179.35	179.10	901.68	174.30	173.70	900.72	175.68	**173.50**	485.47	**174.05**	173.60	706.13
datD3	153.40	152.90	902.33	147.85	147.20	900.82	148.80	**146.20**	749.28	**146.68**	**146.20**	1069.98
datD4	122.35	121.90	902.35	117.75	117.10	901.14	117.22	**115.20**	1138.28	**115.42**	**115.20**	1518.80

Table 3: Comparison of the performances of MMAS, ACS, DQGA and ILS on datasets in groups C and D. Ant algorithms use a simple 2-opt.

per instance instead of ten, these measures are not good enough to ensure an adequate comparison of C (the average solution cost). Additionally, with eight more runs it is probable that C* would be a bit lower, but certainly not higher. Thus, the only uses of this comparison are to show that ACS is close to state-of-the-art performances, but MMAS is not, and to benchmark the impact of improvements in later sections. A complete benchmark will be done on the final version of our algorithm.

The superiority of ACS to MMAS is not hard to explain. If one refers to figure 9, we see that MMAS overtakes ACS after about 9000 seconds of computation time, which is far more than the 900 seconds allowed here. Additionally, as the SSP is a more complex problem, reaching an equivalent number of iterations would take even more time. It is possible that after a way longer run time that MMAS eventually outperform ACS. For a reasonable running time it is unsurprising that ACS is better considering its aggressive construction behaviour.

7.2 Tuning and improving Ant Colony System

In view of the first computational results, we decided to stick with Ant Colony System from now on. We chose to focus on this algorithm obviously because of its performance results but also because of the absence of stagnation behaviour, which is hard to cope with in the case of the SSP. In this section, we will investigate a set of possible improvements to the simple ACS implemented in the previous section by adding or removing different mechanisms to the algorithm and by fitting some parameters.

Although it might seem unnecessarily detailed, we chose to take the step-by-step path to tune the algorithm. With more time and for an actual research paper, a fine-tuning software for heuristics such as CALIBRA (Adenso-Díaz & Laguna, 2006) would have been a better choice than an experimental tuning, or than simply using reported good values (Sörensen, 2012).

Dataset	ACS + 2-opt			ACS + 2.75-opt			DQGA			ILS		
	C	C*	T	C	C*	T	C	C*	T	C	C*	T
datC1	98.90	98.80	900.14	**98.70**	**98.60**	900.56	99.82	**98.80**	145.29	99.09	98.90	99.47
datC2	82.80	82.60	900.18	**82.65**	82.60	900.75	83.74	**82.50**	196.09	82.82	**82.50**	137.08
datC3	67.20	66.90	900.19	67.00	66.90	900.91	67.82	**66.60**	220.32	**66.78**	**66.60**	172.59
datC4	51.70	51.60	900.34	51.55	51.50	901.02	52.10	**51.20**	144.89	**51.47**	51.30	137.79
datD1	198.05	197.60	900.52	**197.35**	**197.10**	902.27	200.16	198.00	213.43	198.36	198.00	488.66
datD2	174.30	173.70	900.72	**173.30**	**172.70**	902.54	175.68	173.50	485.47	174.05	173.60	706.13
datD3	147.85	147.20	900.82	147.00	146.60	902.89	148.80	**146.20**	749.28	**146.68**	**146.20**	1069.98
datD4	117.75	117.10	901.14	116.85	116.30	903.35	117.22	**115.20**	1138.28	**115.42**	**115.20**	1518.80

Table 4: Comparison of the performances of ACS with *2-opt*, ACS with *2.75-opt*, DQGA and ILS on dataset groups C and D.

7.2.1 Using a stronger local search

First, we will use a stronger local search as advised in Dorigo and Stützle (2004). The larger local search that will replace *2-opt* is the *2.75-opt*, that is the *2.5-opt* with the *job swap move* (see section 4.3.2). We could have explored some other possibilities such as the *1-block grouping* that allegedly allowed the ILS to be so performing (Paiva & Carvalho, 2017), but in practice it explores a way smaller solution space and yielded significantly worse solutions than with a *2-opt*. On the other hand, a *3-opt* had a too high computational complexity: iterations were too long and the colony could not reach the exploitation phase within the allowed time. Computations were made on datasets from groups C and D only, two times per problem and for 15 minutes. Results in table 4 show that this improvement allowed to outperform ILS and DQGA on instances with larger magazines than before and greatly improved the quality of the solutions for the hardest datasets (D3 and D4).

7.2.2 Setting $\beta = 0$

Eliminating the heuristic information was found to improve the solution quality returned by ACS for the TSP (Dorigo & Stützle, 2004). We investigated whether this conclusion is valid for the SSP as well. To save time, computation was only made on dataset D4 since it contains the hardest problems. Again, the algorithm ran for 15 minutes twice on each instance. The results (in table 5) compare the results with $\beta = 2$ and show that this is indeed probably a good improvement. We kept the *2.75-opt* for the local search. The explanation is that the ants are no longer biased by this myopic information and instead rely only on the pheromone trail which constitutes the intelligence of the algorithm. The bias created by the heuristic information coupled with the *pseudo-random proportional rule* constrained the ants to choose more often the obvious but myopic moves. Without this bias, the local evaporation rule can ensure that the ants keep exploring the solution

Dataset	Beta = 2		Beta = 0	
	C	C*	C	C*
DatD4	116.85	116.30	**116.2**	**115.6**

Table 5: Comparison of the performances of ACS with ($\beta = 2$) and without ($\beta = 0$) using heuristic information on dataset D4. $q_0 = 0.98$.

space. During the first iteration, 10 solutions are quasi-purely randomly generated and locally optimized, resulting in a strong initial exploration phase. This also reduces the complexity of the construction procedure from $O(MN^3)$ to $O(N)$ because the *pseudo-random proportional rule* (equation 6.7) does not require to compute $\eta_{ij}\ \forall i, j$ anymore:

$$j = \begin{cases} argmax_{l \in N_i^k}\{\tau_{il}\} & if\ q \leq q_0 \\ J(0) & otherwise \end{cases} \tag{7.2}$$

7.2.3 Setting a lower q_0

To increase exploration, we tried the values 0.9 and 0.8 for q_0. Since removing the heuristic information proved to be a great improvement, we decided to benchmark the different values for q_0 with and without the heuristic information, to make sure that $\beta = 0$ is a good idea.

While confirming the dominance of not using heuristic information, the results in table 6 indicate that 0.9 is a better choice than 0.98 for q_0 in either case. The poor results of 0.8 show that it is probably not interesting to try with lower values. To avoid overfitting on dataset D4, we decided not to push the search of a perfect q_0 any further. The point is just to show that ACS for the SSP needs a bit more randomness than for the TSP although a high value is still a good choice.

Dataset	q0	Beta = 2		Beta = 0	
		C	C*	C	C*
DatD4	0.98	116.85	116.30	116.2	115.6
	0.9	116.50	116.10	**115.9**	**115.4**
	0.8	117.40	116.90	116.2	115.8

Table 6: Comparison of the performances of ACS with three different values for q_0, with ($\beta = 2$) and without ($\beta = 0$) heuristic information.

Dataset	ACS			DQGA			ILS		
	C	C*	T	C	C*	T	C	C*	T
datA1	**12.50**	**12.50**	0.88	**12.50**	**12.50**	0.17	**12.50**	**12.50**	0.09
datA2	**10.80**	**10.80**	0.77	**10.80**	**10.80**	0.17	**10.80**	**10.80**	0.09
datA3	**10.10**	**10.10**	0.16	**10.10**	**10.10**	0.19	**10.10**	**10.10**	0.05
datA4	**10.00**	**10.00**	0.00	**10.00**	**10.00**	0.22	**10.00**	**10.00**	0.04
datB1	**26.50**	**26.50**	3.49	26.72	**26.50**	1.62	**26.50**	**26.50**	1.09
datB2	**21.70**	**21.70**	4.53	21.74	**21.70**	2.12	**21.70**	**21.70**	1.38
datB3	**19.70**	**19.70**	2.66	19.76	**19.70**	1.39	**19.70**	**19.70**	1.12
datB4	**19.20**	**19.20**	0.03	**19.20**	**19.20**	0.79	**19.20**	**19.20**	0.71
datC1	**98.83**	**98.50**	100.40	99.82	98.80	145.29	99.09	98.90	99.47
datC2	**82.63**	**82.40**	140.47	83.74	82.50	196.09	82.82	82.50	137.08
datC3	67.00	**66.60**	170.34	67.82	**66.60**	220.32	**66.78**	**66.60**	172.59
datC4	51.65	51.30	140.42	52.10	**51.20**	144.89	**51.47**	51.30	137.79
datD1	**197.11**	**196.50**	525.50	200.16	198.00	213.43	198.36	198.00	488.66
datD2	**172.93**	**172.20**	711.77	175.68	173.50	485.47	174.05	173.60	706.13
datD3	**146.31**	**145.40**	1072.23	148.80	146.20	749.28	146.68	146.20	1069.98
datD4	115.82	**114.50**	1502.43	117.22	115.20	1138.28	**115.42**	115.20	1518.80

Table 7: Benchmark of the final version of ACS with parameters $\alpha = 1,\ \beta = 0,\ \rho = 0.1,\ \xi = 0.1,\ q_0 = 0.9,\ nAnts = 10$ and using the *2.75-opt* local search.

7.3 Computational Results

With those new parameter values and the *2.75-opt*, we decided to stop looking for small improvements that might only bring a contribution to the performances because of overfitting. However, there are several opportunities for improvement that will be proposed in the conclusion of this thesis. The final benchmark will aim to compare our definitive version of Ant Colony System with the existing state-of-the-art algorithms: the Iterated Local Search (Paiva & Carvalho, 2017) and the Dynamic Q-learning Genetic Algorithm (Ahmadi et al., 2018). To do so, we decided to set the stopping criterion of ACS (the maximum time allowed) to approximatively the same as ILS. Since Ant Algorithms can run for an infinite time, we decided that for a fair comparison, fixing the time accordingly was the only possibility. Although the programming language, the hardware and the programming skills also play a significant role in the performances. Problems from groups A and B were given a limit of 250 iterations instead. All problems were solved 10 times. As a reminder, the parameters are $\alpha = 1,\ \beta = 0,\ \rho = 0.1,\ \xi = 0.1,\ q_0 = 0.9,\ K(\#ants) = 10$. The Java code, an executable of the program and a spreadsheet containing the output of our computations are available on a drive at the following URL: `https://tinyurl.com/acs-ssp`.

The results are available in table 7. Our Ant Colony System was able to outperform both algorithms in most of the datasets. It found new best known solutions for all datasets

except for C3 and C4 for which we cannot know if it did. It appears clearly that Ant Colony System is a strong alternative to existing solving methods for the Job Sequencing and Tool Switching Problem.

One may argue that it would achieve more constant results with a clearly defined stopping criterion. Further investigations are needed to define a number of iterations at which we observe a less variable quality of output. That iteration limit may be fixed for all problems or a linear combination of the parameters M, N and C, for example. The stopping criterion used here is only relevant for benchmarking purposes. Indeed, in dataset C4, the allowed time is shorter than that of C3 even though C4 contains harder problems. Both benchmark algorithms (ILS and DQGA) seemed to have needed less time to solve the problems in this dataset. ACS on the other hand had the time to perform around 160 iterations on dataset C4 but 250 on dataset C3. This results in performances that are not really constant. Hence the need of a well defined stopping criterion.

We believe that the success of Ant Colony System is mainly due to its learning capabilities and the strong local search we used. The fact that Ant Colony algorithms are *model-based* (see introduction of chapter 6) is a strong advantage over algorithms that do not learn. ACS is an artificial swarm intelligence: it learns how to tackle the problem by making a good use of the accumulated experience.

The Iterated Local Search of Paiva and Carvalho (2017) is a very simple metaheuristic that does not learn at all. It is just random perturbations followed by a local re-optimization in the hope of landing in a better region of the solution space.

Dynamic Q-learning GA (Ahmadi et al., 2018) works like any genetic algorithm, it keeps the best solutions from a generation and the cross-over and mutations allow to explore the solution space around them. But this is not a learning: the sequences in one generation are only random crossovers of the previous one, no memory whatsoever is kept from past experience to construct the new solutions. The mutations are also purely randomized. Even if it is presented by the authors as a reinforcement learning component, the "Q-learning" in this algorithm is anecdotic in the sense that it does not learn about the problem itself.

Finally, our *2.75-opt*, the local search used to locally improve the solutions of the ants, explores a larger space than other heuristics in the literature. This makes sure that without the heuristic information, η_{ij}, the solutions constructed do not miss out on good pairs of jobs because of the randomness of the ants that are only following the pheromone trail. Unlike the heuristic information, a wide local search is not myopic. By trying different perturbations of a solution, it is more likely to discover new good ideas. This is especially effective when the capacity of the magazine is high because the flexibility offered by the additional slots requires a more strategic job sequencing to solve the problem.

7.4 Summary

We implemented a Min Max Ant System and an Ant Colony System for solving the SSP. The ACO algorithms we propose for the SSP include some customized features that are relevant for this specific problem. We observed that ACS is far superior to MMAS for this specific problem. Then, we improved our ACS by using a stronger local search and by manually tuning some parameters. After a final benchmark, we concluded that our ACS outperforms the state-of-the-art metaheuristics for the SSP on most datasets. It found new best known solutions on almost every non-trivial dataset.

Chapter 8

Conclusion and directions for further research

In this thesis, we investigated the Job Sequencing and Tool Switching Problem, a well-known combinatorial optimization problem arising in task sequencing for flexible manufacturing systems. We address the problem using an Ant Colony Optimization algorithm called *Ant Colony System* that we customized to suit its specific needs.

After having stated the problem and described its known properties, we reviewed the existing methods to solve the problem by adopting a critical point of view on the presented algorithms and results. Because the problem is intrinsically complex, classic exact solution approaches proved to be rather ineffective. As a consequence, most of the research on the problem prefer a heuristic approach coupled with a metaheuristic strategy to find good solutions. We observed that the existing local searches for the SSP were either weak or complex. Thus, we proposed a local search previously unused for the SSP inspired from the famous *2.5-opt* for the Travelling Salesman Problem.

Then a review on Ant Colony Optimization algorithms allowed us to discover this swarm intelligence class of metaheuristics and how it is implemented for solving the TSP, a problem that shares a common backbone with the SSP. We compared the performances of two Ant Colony algorithms to solve the SSP, we selected the best of them, Ant Colony System, and gradually improved it. It eventually outperformed state of the art metaheuristics in the majority of the considered test datasets.

As further research on Ant Colony Optimization for the SSP, we would indicate to investigate the following aspects. First, the use of a more performant programming language such as C, Julia or C++ would allow the colony to generate more iterations of ants in a given time. Second, using a multicolony parallel implementation instead of sequentially generating colonies and retrieving the best solution may help the colonies that get rapidly stuck in bad regions to get back on good tracks thanks to the others that found

a good solution. A cooperation between them might allow the bad colonies to re-bounce with the information they received. Third, we would consider hybridization using more sophisticated local searches than *2.75-opt*, such as a *tabu search*. Finally, a more systematic tuning of the parameters should be performed using a formal methodology instead of a manual search.

Bibliography

Adenso-Díaz, B., & Laguna, M. (2006). Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, *54*(1), 99-114.

Ahmadi, E., Goldengorin, B., Süer, G. A., & Mosadegh, H. (2018). A hybrid method of 2-tsp and novel learning-based ga for job sequencing and tool switching problem. *Applied Soft Computing*. doi: https://doi.org/10.1016/j.asoc.2017.12.045

Alba, E. (2005). *Parallel metaheuristics: a new class of algorithms* (Vol. 47). John Wiley & Sons.

Al-Fawzan, M., & Al-Sultan, K. (2002). A tabu search based algorithm for minimizing the number of tool switches on a flexible machine. *Computers & Industrial Engineering*, *44*, 35-47.

Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2008). *A memetic algorithm for the tool switching problem.*

Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2011). Memetic cooperative models for the tool switching problem. *Memetic Comp.*, *3*, 199-216.

Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2012). Solving the tool switching problem with memetic algorithms. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, *26*(02), 221-235. doi: 10.1017/s089006041100014x

Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2013). Cross entropy-based memetic algorithms: An application study over the tool switching problem. *International Journal of Computational Intelligence Systems*, *6*(3), 559-584.

Balakrishnan, N., & Chakravarty, A. K. (2001). Opportunistic retooling of a flexible machine subject to failure. *Naval Research Logistic*, *48*(1), 79-97.

Bard, J. F. (1988). A heuristic for minimizing the number of tool switches on a flexible machine. *IIE Transactions*, *20*(4), 382-391. doi: 10.1080/07408178808966195

Bean, J. C. (1994). Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, *6*(2), 154-180.

Blażewicz, J., Finke, G., Haupt, R., & Schmidt, G. (1988). New trends in machine scheduling. *European Journal of Operational Research*, *37*(3), 303-317.

Blum, C. (2005). Beam-aco for simple assembly line balanci. *Computers & Operational*

Research, *32*(6), 1565-1591.

Blum, C. (2008). Beam-aco for simple assembly line balancing. *INFORMS Journal on Computing*, *20*(4), 618-627.

Blum, C., Roli, A., & Dorigo, M. (2001). Hc–aco: The hyper-cube framework for ant colony optimization. In P. o. MIC'2001 (Ed.), *4th metaheuristics international conference* (Vol. 2, p. 399-403).

Bullnheimer, B., Hartl, R. F., & Strauss, C. (1999). A new rank-based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics*, *7*(1), 25-38.

Burger, A., Jacobs, C., van Vuuren, J., & Visagie, S. (2015, 04). Scheduling multi-colour print jobs with sequence-dependent setup times. *Journal of Scheduling*, *18*, 131-145.

Catanzaro, D., Gouveia, L., & Labbé, M. (2015). Improved integer linear programming formulations for the job sequencing and tool switching problem. *European Journal of Operational Research*, *244*(3), 766-777. doi: 10.1016/j.ejor.2015.02.018

Chaves, A. A., Lorena, L. A. N., Senne, E. L. F., & Resende, M. G. C. (2016). Hybrid method with cs and brkga applied to the minimization of tool switches problem. *Computers & Operations Research*, *67*, 174-183.

Chaves, A. A., Senne, E. L. F., & Yanasse, H. H. (2012). Uma nova heurística para o problema de minimização de trocas de ferramentas. *Gestão & Produção*, *19*(1), 17-30.

Crama, Y., Kolen, A. W., & Oerlemans, A. G. (1994). Minimizing the number of tool switches on a flexible machine. *International Journal of Flexible Manufacturing Systems*, *6*, 33-54.

Crama, Y., Moonen, L. S., Spieksma, F. C. R., & Talloen, E. (2007). The tool switching problem revisited. *European Journal of Operational Research*, *182*(2), 952-957. doi: 10.1016/j.ejor.2006.07.028

Croes, G. A. (1958). A method for solving traveling salesman problems. *Operations Research*, *6*, 791-812.

Djellab, H., Djellab, K., & Gourgand, M. (2000). A new heuristic based on a hypergraph representation for the tool switching problem. *International Journal of Production Economics*, *64*(1-3), 165-176.

Dorigo, M. (1992). *Optimization, learning and natural algorithms [in italian].* (Unpublished doctoral dissertation).

Dorigo, M., & Gambardella, L. M. (1997a). Ant colonies for the traveling salesman problem. *BioSystems*, *43*(2), 73-81.

Dorigo, M., & Gambardella, L. M. (1997b). Ant colony system: A cooperative learning

approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, *1*(1), 53-66.

Dorigo, M., & Stützle, T. (2004). *Ant colony optimization.* Cambridge: MIT Press.

Finke, G., & Kusiak, A. (1987). Models for the process planning problem in flexible manufacturing systems. *The International Journal of Advanced Manufacturing Technology*, *2*(2), 3-12.

Gambardella, L. M. (2015). *Coupling ant colony system with local search* (Unpublished doctoral dissertation).

Gambardella, L. M., & Dorigo, M. (1996). Solving symmetric and asymmetric tsps by ant colonies. In T. Baeck, T. Fukuda, & Z. Michalewicz (Eds.), *Proceedings of the 1996 ieee international conference on evolutionary computation* (p. 622–627). Piscataway, NJ: IEEE Press.

Garey, M. R., & Johnson, D. S. (2002). *Computers and intractability* (Vol. 29). wh freeman New York.

Gendreau, M., Hertz, A., & Laporte, G. (1992). New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, *40*(6), 1086-1094.

Gendreau, M., & Potvin, J.-Y. (2010). *handbook of metaheuristics* (2nd ed., Vol. 146). Stanford University, CA, USA: Springer. doi: 10.1007/978-1-4419-1665-5

Ghiani, G., Grieco, A., & Guerriero, E. (2007).

Ghiani, G., Grieco, A., & Guerriero, E. (2010). Solving the job sequencing and tool switching problem as a nonlinear least cost hamiltonian cycle problem. *Networks*, *55*(4), 379-385. doi: 10.1002/net.20341

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, *13*(5), 533–549.

Golden, B., & Stewart, W. (1985). Empirical analysis of heuristics. In E. Lawler, J. Lenstra, A. R. Kan, & D. Shmoys (Eds.), *The traveling salesman problem* (p. 207-249). Chichester, U.K.: John Wiley & Sons.

Gonçalves, J. F., & Resende, M. G. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, *17*, 487–525.

Gray, A., Seidmann, A., & Stecke, K. (1988). *Tool management in automated manufacturing: Operational issues and decision problems* [Unpublished Work]. Rochester, New York.

Grötschel, M., & Padberg, M. (1985). Polyhedral theory. In E. Lawler, J. Lenstra, A. RinnooyKan, & S. D.B. (Eds.), *The traveling salesman problem: A guided tour of combinatorial optimization* (p. 689-694). Wiley, Chichester, UK.

Hadji, R., Rahoual, M., Talbi, E., & Bachelet, V. (2000). Ant colonies for the set covering problem. In M. Dorigo, M. Middendorf, & T. Stützle (Eds.), *Abstract proceedings of ants 2000—from ant colonies to artificial ants: Second international workshop on ant algorithms* (p. 63-66). Brussels, Université Libre de Bruxelles..

Helsgaun, K. (2009). General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, *1*(2), 119-163.

Hertz, A., Laporte, G., Mittaz, M., & Stecke, K. E. (1998). Heuristics for minimizing tool switches when scheduling part types on a flexible machine. *IIE Transactions*, *30*, 689–694.

Hoffman, A. J., Kolen, A. W. J., & Sakarovitch, M. (1985). Totally-balanced and greedy matrices. *Journal on Matrix Analysis and Applications*, *6*(4).

Jain, S., Johnson, M., & Safai, F. (1996). Implementing setup optimization on the shop floor. *Operations Research*, *43*, 843-851.

Konak, A., & Kulturel-Konak, S. (2007). *ant colony optimization for the minimization of the number of tool switching instants in flexible manufacturing systems* [Unpublished Work].

Konak, A., Kulturel-Konak, S., & Azizoğlu, M. (2008). Minimizing the number of tool switching instants in flexible manufacturing systems. *International Journal of Production Economics*, *116*(2), 298-307. doi: 10.1016/j.ijpe.2008.09.001

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, *7*, 48-50.

Laporte, G., Salazar-Gonzalez, J. J., & Semet, F. (2004). Exact algorithms for the job sequencing and tool switching problem. *IIE Transactions*, *36*(1), 37-45. doi: 10.1080/07408170490257871

Lawler, E. L., & Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations research*, *14*(4), 699–719.

Maniezzo, V. (1999). Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. *INFORMS Journal on Computing*, *11*(4), 358–369.

Matzliach, B., & Tzur, M. (2000). Storage management of items in two levels of availability. *European Journal of Operational Research*, *121*, 363-379.

McGeoch, L., & Sleator, D. (1994). A strongly competitive randomized paging algorithm. *Algorithmica*, *6*, 816-825.

Meyer, B., & Ernst, A. (2004). Integrating aco and constraint propagation. In M. Dorigo, C. Blum, L. M. Gambardella, F. Mondada, & T. Stützle (Eds.), *Ant colony optimization and swarm intelligence, 4th international workshop, ants 2004. lecture notes in computer science* (Vol. 3172, p. 166–177). Springer, Berlin.

Nesterov, Y., & Nemirovskii, A. (1994). *Interior-point polynomial algorithms in convex programming* (Vol. 13). Siam.

Oliveira, A., Chaves, A., & Lorena, L. (2013). Clustering search. *Pesquisa Operacional*, *33*(1), 105-121.

Paiva, G. S., & Carvalho, M. (2017). Improved heuristic algorithms for the job sequencing and tool switching problem. *Computers and Operations Research*, *88*, 208-219.

Pedemonte, M., Nesmachnow, S., & Cancela, H. (2011). A survey on parallel ant colony optimization. *Applied Soft Computing*, *11*(8), 5181–5197.

Privault, C., & Finke, G. (1995). Modelling a tool switching problem on a single nc-machine. *Journal of Intelligent Manufacturing*, *6*, 87-94.

Privault, C., & Finke, G. (2000). k-server problems with bulk requests: an application to tool switching in manufacturing. *Annals of Operations Research*, *96*(1), 255-269.

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, *177*(3), 2033–2049.

Shmygelska, A., Aguirre-Hernandez, R., & Hoos, H. H. (2002). An ant colony optimization algorithm for the 2d hp protein folding problem. In *International workshop on ant algorithms* (p. 40-52).

Sörensen, K. (2012). Metaheuristics - the metaphor exposed. *International Transactions in Operational Research.*(September). doi: 10.1111/itor.12001

Stecke, K. (1983). Formulation and solution of nonlinear integer production planning problems for flexible manufacturing systems. *Management Science*, *29*, 273–288.

Stützle, T. (1999). *Local search algorithms for combinatorial problems: Analysis, improvements, and new applications* (Unpublished doctoral dissertation).

Stützle, T., & Hoos, H. H. (1997). The max-min ant system and local search for the traveling salesman problem. In T. Bäck, Z. Michalewicz, & X. Yao (Eds.), *Proceedings of the 1997 ieee international conference on evolutionary computation* (p. 309-314). Piscataway, NJ: IEEE Press.

Stützle, T., & Hoos, H. H. (2000). Max-min ant system. *Future Generation Computer Systems*, *16*(8), 889-914.

Tang, C., & Denardo, E. (1988a). Models arising from a flexible manufacturing machine, part i: Minimization of the number of tool switches. *Operations Research*, *36*(5), 767-777.

Tang, C., & Denardo, E. (1988b). Models arising from flexible manufacturing machine, part ii: Minimization of the number of switching instants. *Operations Research*, *36*(5), 778-784.

Wiesemann, W., & Stützle, T. (2006). Iterated ants: an experimental study for the

quadratic assignment problem. In M. Dorigo, L. Gambardella, M. Birattari, A. Martinoli, R. Poli, & T. Stützle (Eds.), *Ant colony optimization and swarm intelligence: 5th internationalworkshop, ants 2006. lecture notes in computer science* (Vol. 4150, p. 179-190). Springer, Berlin.

Zhou, B.-H., Xi, L.-F., & Cao, Y.-S. (2005). A beam-search-based algorithm for the tool switching problem on a flexible machine. *Int J Adv Manuf Techno*, *25*(9-10), 876–882.