A comparison of (meta-)heuristical approaches and constraint programming for solving the job shop scheduling problem

Pascal Wagener, Yann Chevalaz

07.02.2023

Module: Projektseminar Business Analytics

[redacted]

Winter Semester 2022/2023

Production planning is one the most crucial fields of research in modern industry, as designing methods to find the best possible solution to run specific jobs under given constraints is of major importance for production. As for many scheduling problems, there are numerous methods to provide solutions to the Job shop Scheduling Problem (JSP). Scheduling problems have already been thoroughly examined from an academic and scientific stance in the past few decades, yet new challenging scheduling problems arise constantly from novel practical application domains. As various methods are developed and computational means become vastly accessible, it is of great interest to evaluate the most efficient solving methods for given problems. For this reason, the present work aims at solving the JSP using both heuristical approaches and a state of the art constraint programming solver. In that manner, performance of said methods can be compared for different input parameters, helping to highlight advantages of using either methods. At first, scheduling problems and specifically the JSP will be looked into, following which, an algorithm for developing a first solution for JSP instances will be presented. Additionally, underlying concepts of heuristical approaches shall be investigated before effectively solving the JSP both through a (meta-)heuristical approach and with the help of constraint programming. The performance of those methods with various adjustments to the meta-heuristic for several instances shall be evaluated and discussed.

Contents

List of Figures

List of Tables

Glossary

COP Constraint Optimization Problem 21

CSP Constraint Satisfaction Problem 21

FJSP Flexible Job shop Scheduling Problem 6

JSP Job shop Scheduling Problem 2

MpFJSP Multi-plants-based multi-resource Flexible Job shop Scheduling Problem 6

MrFJSP Multi-resource Flexible Job shop Scheduling Problem 6

MRI Mean Relative Improvement 23

PMS Parallel Machine Scheduling 3

SCM Supply Chain Management 1

SMS Single Machine Scheduling 3

1 Introduction

1.1 Production planning fundamentals

Production planning revolves within the world of Supply Chain Management (SCM). A general definition of SCM consists of viewing it as being ”the process of planning, implementing, and controlling the supply chain operations in an efficient way” [KP19]. As such, production planning plays a central role in SCM. The first implementations of planning and scheduling methods date back to the early 1940s in the oil industry, already before the first computers emerged. However it is around the 1980s that production planning gained broader interest and is since then at the very core of SCM [KP19]. Production planning can be defined as ”the planning of acquisition of resources and raw materials, and the planning of production activities which require the transformation of raw materials into finished products, thereby meeting the customers demand in the most efficient and economic way possible” [PW06]. In that sense, production planning could be viewed as the planning of future production given three main aspects. Firstly, the resources available: such resources can be man, machine or money for example. Secondly, the amount of raw material available and needed to create the products. Thirdly, it is important to consider the time horizon to be finite, so as to be able to divide time in specific time units. As such, the planning and scheduling of production focuses on determining the most efficient allocation of limited resources, over a limited time, between competing activities [KP19]. There are two main aspects to consider when it comes to production planning: customer satisfaction and profit maximization or cost minimization. The aim is to find the optimal trade-off between the economic objective and satisfying a given customer demand [RRZ$^+$19]. In other word, producing enough to satisfy clients at a minimal cost. To achieve this goal, there are several time scales which decision makers can act upon. Broadly considered, these three decision-making levels are the following: the strategic level, the tactical level and the operational level [KP19]. These decision horizons are summarized in Figure 1 below.

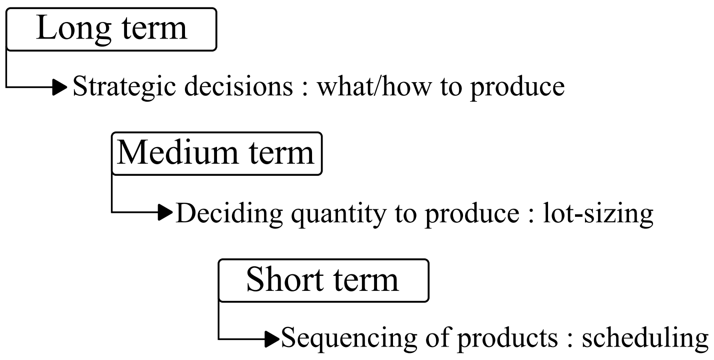
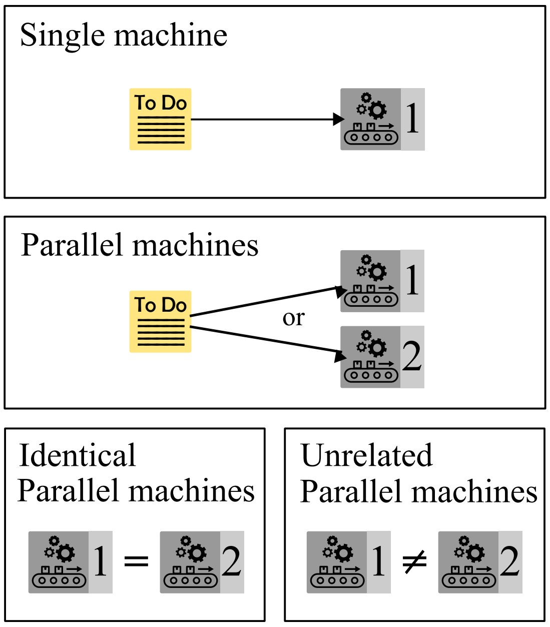

Long term
Strategic decisions : what/how to produce
Medium term
Deciding quantity to produce : lot-sizing
Short term
Sequencing of products : scheduling

Figure 1: Decision horizons in production planning

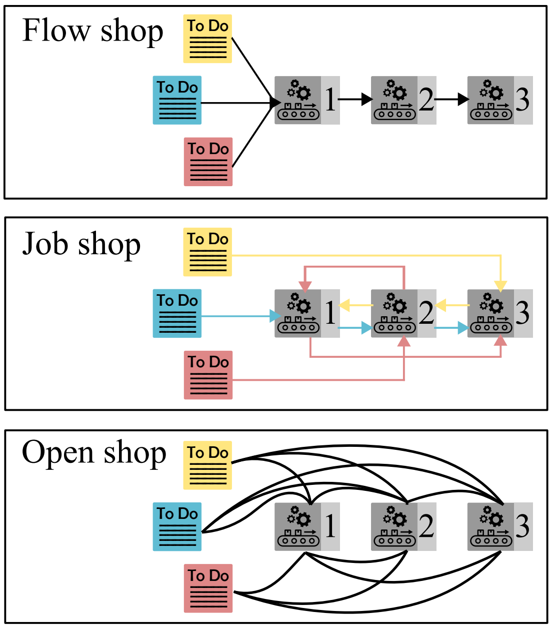
The strategic level is primarily focused on the aspects pertaining to the design of the supply chain. This can also be called long term planning. This level of planning focuses on making strategic decisions. Such decisions involve, among other things, determining which products should be considered for production, the processes which need to be implemented in order for the product to become produceable, and the choice of equipment required for production. As such, on the strategic level, decision makers determine what to produce, and how to produce it. The tactical level involves medium-term planning decisions. The main decision in this level is that of deciding the quantity of products that should be produced in a specific time unit, that is, lot-sizing. In that sense, lot-sizing is neither an activity that can be planned on rather longer terms, nor can it be done within immediate time. The operational level contains decisions dealing with short term problems. The major problem of this type is the scheduling problem which aims at determining the sequence of production for given items over a specific time period. As the present work will focus on solving a scheduling problem, solely the operational level of decision making in production planning will be dealt with.

1.2 Scheduling problems

This section will aim at providing a broad overview of scheduling activities. Scheduling will be defined and the different categories of scheduling problems will be examined. Yamada and Nakano refer to scheduling as the "allocation of shared resources over time to competing activities" [YN97]. The usual scheduling problem considers resources to be production machines while activities are represented as jobs, where each machine can only process one job. Lin and Huang propose a division of scheduling activities into four main categories pictured in Figure 2a [LH21].



(a) Single operation scheduling



(b) Multiple operation scheduling

Figure 2: Classification of scheduling problems

Single Machine Scheduling (SMS) refers to a production environment where only a single operation needs to be treated. All the jobs required for that operation are treated on one machine. SMS problems play a rather fundamental role in developing and understanding more sophisticated problems as they can be seen as ground laying problems. Contrarily to their multi-machines counterparts, several SMS problems can be optimally solved within polynomial time. The problem of single machine scheduling has been studied for several objective functions and constraints such as minimizing energy consumption or minimizing the makespan for example. The makespan is defined as ”the completion time of the job that will be completed last” [JP19], and is the objective value along which scheduling problems tend to be optimized in most cases. Parallel Machines Scheduling (PMS) refers to a model in which several machines with similar function are available. As such, a job could either be treated on one machine or another, it is not required for a job to be treated on every machine. It is important to note that these machines can work simultaneously and independently. Depending on whether the machines working in parallel are the same or not, PMS models can be separated into identical parallel machines and unrelated parallel machines. In the first case, the machines considered are homogeneous in their characteristics and each machine takes the same time to treat a job. As such, the processing time is exactly the same on each machine. In the second case, that of unrelated parallel machines however, these can differ and processing time can vary accordingly. The most frequently investigated types of scheduling problems in literature are multiple operation scheduling problems, as these tend to be more relevant for practical situations. Among the multiple operation category, three main forms of problems can be distinguished. Firstly, the flow shop scheduling problem is a problem in which every job should be treated on every machine one and only one time, with the order in which the jobs are treated on the machines being the same for every job. The processing time of each of the jobs can differ from one another. One might consider three different jobs, as shown in Figure 2b, all of which being treated on three distinct machines. In this example, the jobs must first be treated on machine 1, then machine 2, and finally on machine 3. However, as each machine can only process one job at a time, determining the most efficient order in which the jobs are treated on each machine so that the makespan is minimized is of great relevance. However, it appears that most of the time the flow shop problems addressed in literature are so-called permutation flow shops, in which once the sequence in which the jobs go through the first machine is defined, it will be kept for the following machines as well [BEP+19a]. The flow shop scheduling problem is considered a special form of the JSP in which the jobs must be treated in the same order. The JSP makes it possible for jobs to have different orders on the machines, which are to be specified. In the figurative job shop example shown in Figure 2b, all three jobs must go through the three machines in different sequences. If there is no specified order in which jobs must be treated on machines, the problem is considered an open shop, in which every order is possible for every job. In this case as well as in the flow shop and JSPs, all jobs must be treated on all machines. In terms of complexity, open shop problems can be considered the hardest of scheduling problems to solve. Especially for instances with a greater amount of tasks (also referred to as operations), open shop problems cannot be solved. Although the open shop problem can be considered similar to its counterpart the JSP in many aspects, it has been pointed out that while the JSP can nowadays be solved for problem instances reaching up to 100 tasks, many instances featuring less than half as much instances remain unsolved so far for the open shop problem [BEP+19b]. This section did not aim at providing an exhaustive list of existing types of scheduling problem but much rather at providing a general classification of the broadest scheduling problems categories. After introducing single and parallel machines scheduling, flexible, job and open shops, solely the JSP will furtherly be examined in detail.

1.3 The Job shop Scheduling Problem

This section will present a detailed overview of the JSP, analyzing its complexity and some of its variants.

The JSP, as briefly introduced in section 1.2, represents the most basic form of problem for the job shop environment. In the basic JSP, each job to be performed has a given processing order to be respected when going through the machines. The constraints that must be respected for both machines and jobs are as follows [BEP+19c]: firstly, there are no precedence constraints among tasks of different jobs, in other words, no task can be started only if a previous task is done first. Secondly, the tasks cannot be interrupted meaning that if a task has started, it must be finished before another one is treated, as a machine can only treat one job after another. Thirdly, it is impossible for a job to be treated on more than one machine at a time, as such, if a job is treated on a machine, it can only be treated on another other if treatment on the first is finished. The goal of the basic JSP problem is to determine the operation sequence. This must not be confused with machine sequence, which is defined beforehand. The objectives along which the operation sequence is usually optimized are those of minimizing the maximal makespan and or minimizing productions costs.

Formally, the problem can be described using following formulation [BEP+19c] based on [ABZ88]: consider $\mathcal{T} = \{T_0, T_1, ..., T_n\}$ the set of tasks where T_0 and T_n, respectively the first and last task of all jobs, are to be understood as so-called dummy tasks, having each no processing time. Each task T_i has a given processing time p_i. The earliest possible time a task T_i can start is described by t_i and is to be calculated (t_n is therefore the earliest possible time the last task of the last job can be started). Furtherly consider $\mathcal{P}$ the set of m machines and $\mathcal{A}$ the set of task pairs (T_i, T_j) on which lies a constraint of precedence $T_i \prec T_j$. Finally, let $\mathcal{E}_k$ denote the set of all task pairs that must be processed on a machine for each machine P_k, that is, the set of tasks which cannot occur at the same time. The mathematical model for the JSP can then be formulated as follows:

$$\begin{array}{llllll} min & t_n & & & & \\ s.t. & t_j - t_i & \geq & p_i & & \forall\, (T_i, T_j) \in \mathcal{A} & (1.a) \\ & t_j - t_i & \geq & p_i & or \quad t_j - t_i \geq p_i & \forall\, \{T_i, T_j\} \in \mathcal{E}, \forall P_k \in \mathcal{P} & (1.b) \\ & t_i & \geq & 0 & & \forall\, T_i \in \mathcal{T} & (1.c) \end{array} \qquad (1)$$

The aim is to find the minimal earliest possible time for the last task of the last job to start. The first constraint of Equation 1 ($1.a$) ensures that the processing order of tasks within a job is respected. The second constraint ($1.b$) makes sure a machine can solely process one job at a time. The third and last constraint ($1.c$) ensures that all jobs must be processed.

Several types of extensions and variants to the original JSP have been brought to existence by academics in the past decades such that it can be useful to classify some of the most frequently encountered types of JSP in literature. The following classification relies on the work presented in [ZDZ+19]. The authors propose a classification based on the type of structures among which the basic type, the multi-machine type, the multi-resource type, the multi-plant type and finally the smart factory type. Only the first four types of structures will be presented in the present work and are summarized in Table 1. First of all, it is important to note that even for the most basic form of JSP, it is in most of the common cases an NP-hard problem [Lew83]. Accordingly, all the following models, which are extensions of the basic JSP, are also NP-hard in terms of computational complexity. A more advanced as well as more complex problem is the flexible JSP, also referred to as FJSP. The specificity of this problem when compared to its basic counterpart lies in the fact that the choice of machines on which the tasks must be processed is free. As such, the FJSP is a problem in which it is allowed for a task to be processed by any given machine among the available machines in the set of alternative machines [CK16]. The goals of the FJSP are therefore twofold. Firstly, as for the basic JSP, determining the operation sequence. Secondly, assigning operations to suitable machines. This makes the FJSP more complex than the JSP because for each operation, the machine to which it is assigned must be determined. Chaudhry describes the FJSP as comprising two separate subproblems: firstly, a routing problem, dealing with the selection of an adequate machine among the possible ones for each operation, and secondly, a scheduling problem in which, once the assignment of operations to machine has been done, the operation sequence must be determined so as to minimize a given objective [CK16]. Most algorithms developed in literature regarding the solving of scheduling problems focus on the JSP and the FJSP. However, the reality of industrial plants is often more sophisticated. For this reason, two additional models are to be presented here. So far, it has been considered that a machine is considered available as long as it is not currently processing a task. The Multi-resource Flexible Job-shop Scheduling problem (MrFJSP) proposes a more nuanced view through the consideration of several manufacturing resources constraints. As such, it could be possible for a machine to be available for processing a task, but not to be available for scheduling, for other reasons relating to different resources. This model not only considers the characteristics of the FJSP but also inputs and constraints such as plant resource information and layout information for example [ZDZ+19]. In addition to the optimization objectives of the JSP and FJSP, MrFJSP can also be used to aim at minimizing resource transition times, that is, having the least possible time between separate uses of a resource. Finally, the Multi-plants-based multi-resource Flexible Job-shop Scheduling Problem (MpFJSP) presents an extension to the MrFJSP as it adds the possibility of having multiple plants to the model. As multiple plants exist, transportation costs from one plant to another must be taken into account. This model is among the most complex scheduling models as its goal is to find the best possible operation sequence with selected machines, resources and multiple plants. This allows for other optimization objectives than the traditional makespan minimization such as minimizing vehicle mileage for instance.

Model	Complexity	Objective	Goal
Job-shop Scheduling Problem (JSP)	NP problem	Min. makespan/ Min. costs	Operation sequence
Flexible Job-shop Scheduling Problem (FJSP)	NP problem	Min. makespan/ Balance workloads	Operation sequence with selected machines
Multi-resource Flexible Job-shop Scheduling Problem (MrFJSP)	NP problem	Min. makespan/ Balance workloads/ Min. resource transition times	Operation sequence with selected machines and resource
Multi-plants-based multi-resource Flexible Job-shop Scheduling Problem (MpFJSP)	NP problem	Min. makespan/ Min. tardiness/ Min. vehicle mileage	Operation sequence with selected machines, resource and plant

Table 1: Subdivisions of the JSP

There exist several methods to solve the JSP. A broad overview of those methods is proposed in Figure 3. Mainly to be differentiated are exact methods and heuristics. Exact methods systematically solve the JSP to optimality while heuristics aim at finding "near-optimal solutions within acceptable running times" [AvLLU94]. An example for an exact method is that of constraint programming solvers which are to be detailed in subsection 3.3. Problem-specific heuristics are those which make use of the specificity of the problem at hand to better solve it. Among those, one can distinguish between constructive methods, which build a solution progressively, and improvement methods, which start with a valid solution and iteratively improve it. An example for the former is the Giffler and Thompson algorithm detailed below in 2.1, a good example for the latter is that of a local search based heuristic, presented in 3.1. As soon as some sort of guidance of the heuristic is introduced, the heuristic is referred to as a meta-heuristic. A proposed strategy to guide local search is that of the tabu search, detailed in subsection 3.2. The present work aims at implementing a tabu search meta-heuristic and comparing it with an exact method that is a CP-solver, as depicted in subsection 3.3.

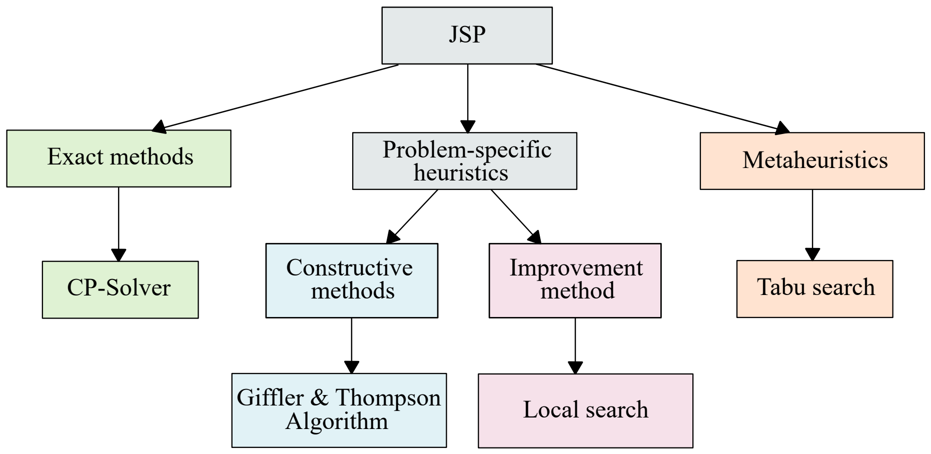


Figure 3: Solving the JSP

2 Preliminary procedures for tabu search

This section will provide detailed information about preliminary work which is required in order to carry out tabu search. Firstly, the Giffler and Thompson algorithm, used to generate a first feasible solutions for the JSP, shall be presented. Then, critical operations and the concept of critical path are to be introduced. Following which neighborhood definitions are to be detailed, and the procedure to approximate the makespan of neighboring solutions will be explained.

2.1 Giffler and Thompson algorithm

A common method applied for solving scheduling problems are priority rules. Using priority rules allows for low time complexity as well as consequent implementation ease. A priority rule is a rule which declares priority of a task over another task on all machines. In that sense, an example for a priority rule would be to process a specific task T_i before another task T_j on all machines. While it would be possible to plainly apply priority rules in the case of a flow shop for instance, doing so for a JSP would not be efficient as it could occur that task T_j could have been started before task T_i on a given machine [JP19]. In order to avoid such situation, Giffler and Thompson developed an algorithm for the JSP which permits to generate all existing active schedules of a JSP. The authors define an active schedule as "a feasible schedule having the property that no operation can be made to start sooner by permissible left shifting" [GT60]. A feasible schedule is a schedule which leads to the required processing of jobs. In allowing for all existing active schedules to be determined, the Giffler and Thompson algorithm can help reducing the potential schedules to consider, to solely those in which no task could have been strated earlier without delaying the starting time of another task. Before detailing the algorithm, some denotations must be defined. Consider that the operation of job i on machine j is denoted by operation (i, j). Let r_{ij} denote the earliest time at which operation (i, j), can start. Let pt_{ij} represent the processing time of operation (i, j), C^* the minimum earliest end time of all available operations, j^* the machine which is to process an operation next, W all available operations and Q the available operations on machine j^*. The Algorithm 1 as implemented for the purpose of the present work is adapted from [LGPI97] and relies on work presented in [BDP96]. The algorithm finds an active schedule in the following manner: firstly it considers a list of all tasks available for scheduling (see line 1). Among those, the algorithm selects the one having the shortest completion time (see line 4). Then, the machine on which this task is to be processed is determined (see line 5). Among the tasks that can be processed on this machine at the current time point, which build the so-called conflict set, the algorithm chooses one accordingly to a priority rule defined beforehand (see lines 7-11). Once this is done, the list of tasks available for scheduling and the earliest possible completion time for a task are updated (see lines 12-14) and the algorithm operates the procedure described above anew. Once all tasks have been scheduled, an active schedule is obtained. This is of great interest as the optimal schedules, if there are any, are part of the active schedules.

Algorithm 1: Giffler and Thompson algorithm

```
Input: Instance
Output: Active schedule
1  Let $W$ contain the first operation of each job
2  $r_{ij} \leftarrow 0,\ \forall (i,j) \in W$
3  while $W \neq \{\}$ do
4      $C^* \leftarrow min\{r_{ij} + pt_{ij},\ \forall (i,j) \in W\}$
5      $j^* \leftarrow$ machine on which minimum is achieved
6      $Q \leftarrow \{\}$
7      forall $(i,j) \in W$ do
8          if $j = j^*$ then
9              if $r_{i,j} < C^*$ then
10                 $Q \leftarrow Q \cup \{(i,j)\}$
11     selectedOperation $\leftarrow$ Select one operation from Q by priority rule
12     $W \leftarrow W \setminus$ selectedOperation
13     $W \leftarrow W \cup$ job successor of selectedOperation
14     Update $r_{ij}$
```

In order to obtain all active schedules, all possible combinations of choices within the conflict set must be considered. While all resulting schedules obtained by using Giffler and Thompson's algorithm are active, they do not necessarily yield results of similar quality. When minimizing the maximal makespan, the choice of priority rule can play a major role on the resulting makespan. For this reason, several priority rules should be tried out when applying Giffler and Thompson's algorithm for a JSP. As a schedule resulting from Giffler and Thompson's algorithm can be used as a starting solution for a heuristical approach, starting with a best possible solution is of great interest. Some of the most commonly encountered priority rules are displayed in Table 2 which relies on an extensive list of priority rules frequently used and mentioned in literature [Hau89].

rule	**signification**	**description**
Rand	Random	Select job at random
FCFS	First come first served	Select first job in the queue
SPT	Shortest processing time	Select job which requires the least time to process
SPT_op	Shortest processing time	Select operation which requires the least time to process
LPT	Longest processing time	Select job which requires the most time to process
LWKR	Least work remaining	Select job with the least work remaining
MWKR	Most work remaining	Select job with the most work remaining

Table 2: Common priority rules for the JSP

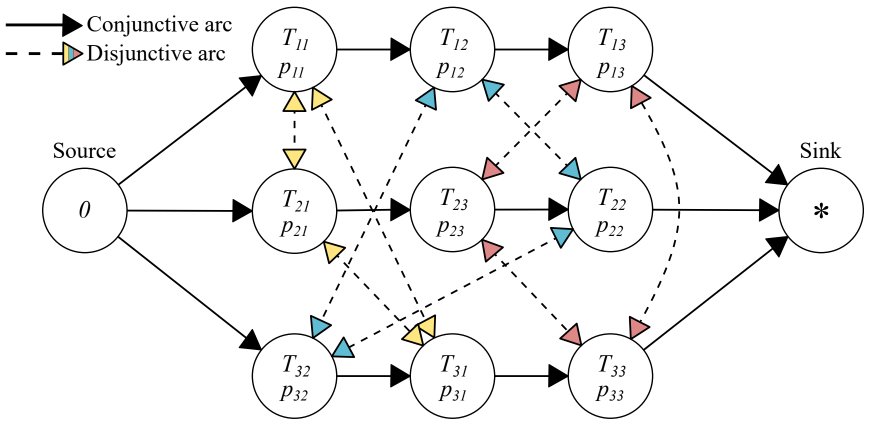
Both the random and FCFS rule are arbitrary rules and usually serve no other purpose than dealing as benchmarks for other priority rules. The former simply states that the job to be selected among the possible ones should be selected at random. The latter considers that if several jobs are waiting to be processed on a machine, they should be processed according to the order in which they arrived. The SPT and LPT rules state that the next job to be selected should be the one which requires respectively the least and the most time to be processed. These rules are not to be confused with the LWKR and MWKR for respectively least and most work remaining rules. Contrarily to SPT and LPT, the LWKR and MWKR rules take the total remaining processing time of a job into consideration and not solely the processing time of a job. When considering priority rules which are best suited in a maximal makespan minimization context for the JSP, it has been shown that the MWKR and SPT rules are of great relevance [SGV12]. Accordingly, these priority rules, along with the random rule for comparison purposes will be applied when using the Giffler and Thompson algorithm in the present work, so as to generate a first feasible solution which shall be used as a starting solution for the upcoming tabu search meta-heuristic.

2.2 Critical operations and path

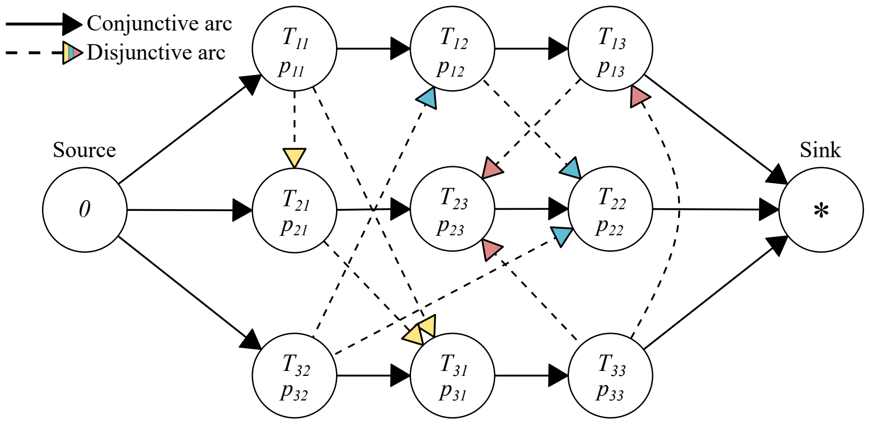
Critical operations are those that cannot be delayed without lengthening the makespan. The critical path is the longest path one can find through a schedule. In that sense, it represents the longest possible time it takes to get from the first job to the last job. There can be more than one critical path and it can also happen for all existing paths in a schedule to be critical paths in extreme cases [UZ12]. In order to facilitate the understanding of critical paths which can be found in a schedule, it can be interesting to represent the JSP through a disjunctive graph $G = (V, C \cup D)$ as shown in Figure 4a below. In this example T_{32} refers to a task of job 3 on machine 2 and p_{32} refers to the processing time the task of job 3 on machine 2. The graph comprises a set of nodes V which represents the tasks of each job on each machine. The dummy start and end tasks are represented as source and sink denoted respectively by 0 and *. Thus, node 0 represents the first task of every job in the schedule and * the last task of every job. C represents the set of conjunctive arcs, which depicts the machine sequence of jobs, also referred to as precedence constraint among tasks of a same job. On the other hand, D comprises the disjunctive arcs, which are used to represent pairs of operations which have to be processed on the same machine. As such, for each pair of tasks $\{T_i, T_j\} \in V$ that must be processed on the same machine, two symmetrically opposed arcs (i, j) and (j, i) exist. The processing times of each task on each machine can be represented in different manners. Some scholars depict the processing time through weights on the arcs [BEP+19c] while others place the weights on the nodes [YN96]. Figure 4a shows a graph in which conjunctive and disjunctive arcs are drawn, the former being directed and the latter undirected. It is the goal of JSP to find in which order the tasks should be processed on the machines, which is equivalent to selecting solely one directed arc among each pair of disjunctive arcs for each machine. In doing so, a directed acyclic graph is generated, as shown in Figure 4b. Such a graph represents a feasible schedule, within which the longest path from source to sink provides the makespan C_{max} of the schedule. Such a path is a critical path and is composed of several critical tasks. A critical task can formally be described as follows [Tai94]:

$$r_i + t_i = C_{max} \tag{2}$$

With every task $i \in V$ and r_i denoting the release date and t_i the length of the tail. The release date is the earliest processing time for a task and the length of the tail represents the length of a longest path from the end of a task to the sink. In other words, if the sum of the earliest starting point of a task and the remaining length of a longest path after this task to the sink is equal to the makespan, then the task is said to be critical. If several critical operations on the same machine are succeeding they form a critical block.



(a) Disjunctive graph



(b) Acyclic graph: a feasible schedule

Figure 4: Disjunctive graph and feasible schedule example

Swapping two operations is of particular interest in order to minimize the makespan and leads to an even more interesting optimization if two critical operations are swapped [BDP96]. The procedure to determine critical operations is to be presented from an algorithmical standpoint below. Algorithm 2 presents the approach implemented as part of the present piece of work in the form of a pseudo code adapted from [Tai94]. Let J_i denote the job of which operation i is part, M_i the machine on which operation i is to be processed, pt_i the processing time of operation i, PJ_i the operation which precedes operation i and belongs to job J_i, and SJ_i inverserly the operation which belongs to job J_i and succeeds i. PM_i represents the operation processed previously to operation i on machine M_i and SM_i the operation processed consequently to operation i on machine M_i. Furthermore r depicts the release time and t the length of the tail (also known as tail time). Let X denote the set of operations for which r can be calculated and Y the set of operations for which t can be calculated, and let K be the list of all critical operations.

Algorithm 2: Find critical operations

```
   Input: $pt, PJ, SJ, PM, SM, C_{max}$
   Output: Critical operations
   /* Calculate r                                                        */
1  $X \leftarrow \{\}$
2  forall $i \in \{operations\}$ do
3      if $PM_i$ and $PJ_i$ do not exist then
4          $X \leftarrow X \cup \{i\}$
5  while $X \neq \{\}$ do
6      $i \leftarrow\in X$
7      $X \leftarrow X \backslash \{i\}$
8      $r_i \leftarrow max\{r_{PM_i} + pt_{PM_i}, r_{PJ_i} + pt_{PJ_i}\}$
9      if $PM_{SJ_i}$ does not exist or $r_{PM_{SJ_i}}$ already calculated then
10         $X \leftarrow X \cup \{SJ_i\}$
11     if $PM_{SM_i}$ does not exist or $r_{PJ_{SM_i}}$ already calculated then
12         $X \leftarrow X \cup \{SM_i\}$
   /* Calculate $t$                                                      */
13 $Y \leftarrow \{\}$
14 forall $i \in \{operations\}$ do
15     if $SM_i$ and $SJ_i$ do not exist then
16         $Y \leftarrow Y \cup \{i\}$
17 while $Y \neq \{\}$ do
18     $i \leftarrow\in Y$
19     $Y \leftarrow Y \backslash \{i\}$
20     $t_i \leftarrow max\{t_{SM_i} + t_{SJ_i}\} + pt_i$
21     if $SM_{PJ_i}$ does not exist or $r_{SM_{PJ_i}}$ already calculated then
22         $Y \leftarrow Y \cup \{PJ_i\}$
23     if $SJ_{PM_i}$ does not exist or $r_{SJ_{PM_i}}$ already calculated then
24         $Y \leftarrow Y \cup \{PM_i\}$
   /* Find critical operations                                           */
25 $K \leftarrow \{\}$
26 forall $i \in \{operations\}$ do
27     if $r_i + t_i = C_{max}$ then
28         $K \leftarrow K \cup \{i\}$
```

The first part of the algorithm relating to the calculation of the release time is to be explained as follows: the algorithm starts by assigning the operations for which r can already be calculated to the empty set X. These are the operations which are to be scheduled at first on a machine and which are the first of a job. In other words, those which have neither machine predecessor nor job predecessor (see lines 2-4). Starting from this set X, an operation i is selected and is thus removed from X. For operation i, r is now to be calculated. For this calculation, the earliest end time of the machine predecessor $r_{PM_i} + pt_{PM_i}$ and the job predecessor $r_{PJ_i} + pt_{PJ_i}$ are to be considered. The release time r results from the maximum of both end times. Afterwards, the algorithm checks whether further operations can be calculated starting from operation i. A such further operation could either be the machine or the job successor of operation i (see lines 9-12). In the first case, one must verify that the predecessor of the machine successor of operation i, namely PM_{SM_i}, does not exist or that its r has already been calculated. If this is the case, then the machine successor of i is to be added to set X. The process functions in similar fashion for the job successor. The algorithm ends as soon as X is empty, that is, when r has been calculated for all operations. The calculation of tail times occurs similarly to that of release times (see lines 13-24). However, in contrast to the calculation of r, the schedule is analyzed from the right (back) to the left (front). Subsequently, for each operation it is to be checked whether $r + t = C_{max}$, that is, whether the operation is a critical one. If so, the operation is added to K (see lines 25-28).

2.3 Neighborhood structures

Identifying the critical operations is of major importance in order to improve the found schedule. There are many possible types of changes that can be made on critical operations so as to transition from a schedule to another one [AvLLU94], these are also known as neighborhood structures. The present paragraph will aim at presenting two rather simple neighborhood definitions frequently encountered in literature. Considering a solution x, let $N(x)$ denote the neighborhood of solution x. As such, if a solution $x' \in N(x)$, then x' is a neighboring solution of x. A first neighborhood definition which served as base for the earlier local search heuristics for solving the JSP, as for example in [Tai94], can be defined as follows:

N_1 : A neighboring solution is found by selecting two succeeding tasks on a given machine which form a disjunctive arc situated on a critical path and replacing this arc by its opposite.

In other words, choosing two tasks which follow each other directly and are to be processed on the same machine, and for which the arc binding these tasks is part of a critical path, then reversing this arc. As such, the arc i, j is replaced by its j, i counterpart. In doing so, the order in which the tasks are processed on the machine is reversed and the graph remains acyclic. The permutation of two successive critical tasks which require the same machine is of great interest as it has been shown that when using this neighborhood structure, the output schedule remains feasible and thus, it is possible to reach an optimal schedule among the feasible ones [AvLLU94]. It is important to note that all possible pairs of succeeding tasks situated on a critical path are considered, and not solely pairs situated on only one critical path. An example procedure for the N_1 neighbordhood is provided in Figure 9 which can be found in the Appendix. As can be seen, a first feasible schedule with makespan C_{max}= 467 is found. By searching for a neighbor solution using neighborhood definition N_1, the processing order of task 2 of job 3 and task 2 of job 2 which are situated on the identified critical path, is reversed. This reversal leads to a feasible schedule with makespan C_{max}= 431, of better length than the one found previously.

As the N_1 neighborhood structure is to be implemented in this work, it is detailed algorithmically below in Algorithm 3. Let swap denote the permutation of two critical tasks. Denote fo as first operation and so as second operation. Consider PM_{fo} the operation processed previously to operation fo and SM_{fo} the operation processed consequently to operation fo. Considering PM_{so} the operation processed previously to operation so and SM_{so} the operation processed consequently to operation so. PM' and SM' respectively denote the actualised PM and SM. For example, for the case of the instance shown in Figure 9 the predecessor of task 2 of job 3 is actualised to task 2 of job 2 (see line 5).

Algorithm 3: N_1 neighborhood definition

```
Input: PM, SM, swap
Output: PM', SM'
1  fo = 1st swap element
2  so = 2nd swap element
3  $PM' \leftarrow PM$
4  $SM' \leftarrow SM$
5  $PM'_{fo} \leftarrow so$
6  $SM'_{fo} \leftarrow SM_{so}$
7  $PM'_{so} \leftarrow PM_{fo}$
8  $SM'_{so} \leftarrow fo$
9  $PM'_{SM_{so}} \leftarrow fo$
10 $SM'_{PM_{fo}} \leftarrow so$
```

As for the second neighborhood structure, denoted N_2, the permutation of tasks occurs not between two successive tasks but between a given critical task and the first or last critical task of the critical block it is part of [BJS94]. A formal definition of N_2 can be provided in the following manner:

N_2: A neighboring solution is found by selecting a critical task within a critical block and placing it either to the rear or to the front of the critical block.

In doing so, either a so-called before candidate schedule is found in the case of a shift to the front of the block, or an after candidate, if the shift occurs to the rear of the critical block. While all possible neigbhoring solutions of a feasible schedule that can be found when applying the N_1 neighborhood structure are also feasible schedules, this is not the case for the N_2 definition. It has been shown that "a set of all before and after candidates may contain infeasible schedules" [YN96]. Although the second neighborhood structure has been shown to deliver results of greater quality than the first one [YN96], the first benefits from a greater ease of implementation and will therefore be used in the present work's heuristical approach to solving the JSP. It is however not to be doubted that an even greater amount of neighborhood definition exist.

2.4 Makespan approximation

The calculation of the approximate makespan of neighboring solutions must occur in every iteration of the tabu search. Following Taillard's depiction, some formal denominations must be introduced in order to detail the aforementioned calculation [Tai94]. Consider C_{max}^{approx} the approximation of the makespan, r^{approx} the approximation of the release time, and t^{approx} the approximation of the tail. The approximation of the makespan of neighboring solutions is then calculated as described in Algorithm 4 below:

```
Algorithm 4: Approximate the makespan
  Input: PM, SM, PJ, SJ, pt, r, t, swap
  Output: $C_{max}^{approx}$
1 fo = 1st swap element
2 so = 2nd swap element
3 $r_{so}^{approx} \leftarrow max(r_{PM_{fo}} + pt_{PM_{fo}},\ r_{PJ_{so}} + pt_{PJ_{so}})$
4 $r_{fo}^{approx} \leftarrow max(r_{so}^{approx} + pt_{so},\ r_{PJ_{fo}} + pt_{PJ_{fo}})$
5 $t_{fo}^{approx} \leftarrow max(t_{SM_{so}}, t_{SJ_{fo}}) + pt_{fo}$
6 $t_{so}^{approx} \leftarrow max(t_{fo}^{approx}, t_{SJ_{so}}) + pt_{so}$
7 $C_{max}^{approx} \leftarrow max(r_{so}^{approx} + t_{so}^{approx},\ r_{fo}^{approx} + t_{fo}^{approx})$
```

With consideration to Equation 2 and Algorithm 4, the after swap makespan of the newly found critical path is then $C_{max}^{approx} = max(r_{so}^{approx} + t_{so}^{approx}, r_{fo}^{approx} + t_{fo}^{approx})$. This approximation allows for the calculation of a lower bound for the makespan. In the case that neither fo nor so are situated on a critical path after the swap, then the actual makespan is greater (and as such, worse) than the approximated one. If the approximated makespan is better than the best neighbor, proceed with the swap and verify with the exact makespan. If the approximated makespan is not better, then do not carry out the swap. By doing so, the running time of the algorithm can be reduced.

3 Solving the JSP

This section puts several approaches to solving the JSP into perspective. Firstly, heuristical methods relying on local search are to be introduced. Especially tabu search will be detailed. Secondly, constraint programming will be investigated and a state of the art solver for combinatorial optimization problems, OR-Tools, will be presented.

3.1 Heuristical approach: local search

The underlying concept of local search methods is that of iterative improvement, as local search algorithms start with an existing solution and progressively find improving solutions. In that sense, these types of algorithms are opposed to constructive algorithms such as the branch-and-bound algorithm for example, which incrementally builds a solution. One can distinguish four main components of local search algorithms [Wat03] namely the state space, the objective function, the move operator and the navigation strategy. Especially the last two components mentioned are of great interest as the first two are given in the problem formulation of the JSP. On the one hand the move operator refers to the neighborhood structure N mentioned in subsection 2.3. This represents the locality definition of local search and defines what is indeed close and what is not. On the other hand, the move operator also provides definition of the connectivity of the search space. In other words, which solutions is possible to attain after a given finite amount of move operations starting from a given initial solution. The first possibility is that the search space is characterized as being connected, in the case that there systematically exists a sequence of moves that leads to an optimal solution starting from a specific solution. The second possibility is that of a fully connected search space, which is provided when there systematically exists a sequence of moves from any given solution to another one, hence, it is possible to attain every solution starting from any solution. As for the navigation strategy, this component refers to the manner in which a neighbor $x' \in N(x)$ is selected. The simplest selection criterion is that of selecting a neighbor $x' \in N(x)$ for which the objective function value $F(x') < F(x)$. Other commonly implemented navigation strategies are those of next-descent and steepest-descent. While the first proposes to order the $N(x)$ neighbors at random and then select a neighbor $x' \in N(x)$ for which $F(x') < F(x)$, the second selects the neighbor x' for which $argmin_{x' \in N(x)} F(x')$. In that sense, steepest-descent selects the neighbor which leads to the greatest objective function decrease compared to the actual solution. These three navigation strategies are greedy strategies, hence, they suffer from the getting stuck at a local optimum problematic. Continuing the search beyond local optima can be done by using meta-heuristics, which reactively adapt the heuristic at hand. Watson defines a meta-heuristic as "a heuristic that dynamically alters the behavior of the core local search heuristics" [Wat03]. In the local search context, the core heuristic would be one of the three previously mentioned greedy descent strategies, and the meta-heuristic a navigation strategy which is activated solely when the greedy descent strategy led to a local optima. However in most cases, the disctinction between core and meta-heuristic is not made. Among the most commonly established meta-heuristics for solving the JSP are tabu search and simulated annealing. While both are of relevance, solely the former will be applied in the present work and thus, shall be presented in detail. A graphical representation of the local search algorithm implemented as part of this work is represented as a flow chart in Figure 10 in the Appendix. Noticeable is that the local search procedure makes use of the makespan approximation precedently introduced and of the steepest-descent procedure.

3.2 Tabu search meta-heuristic

Tabu search steers local search to overcome the problem of getting stuck at a local optimum. Perhaps among the first successful applications of tabu search for the JSP is that of Taillard in the early nineties [Tai94], although tabu search was introduced a few years before by Glover [Glo86]. Tabu search overcomes one of the main problems of other basic local search heuristics such as simulated annealing, namely the memory problem. As local search procedures explore neighboring solutions, it might be possible for a local optimum which has already been visited to be visited again during the search. In order to avoid such a repetition, a form of search memory must be implemented. Tabu search introduces a such adaptive memory which renders cyclical search impossible, since a potential move leading back to a local optimum is added to a tabu list and as such, forbidden. Among the adaptive memory, two forms can be distinguished. Firstly the short term, for simplistic tabu search, and secondly the long term memory, which is additionally used in more advanced tabu search methods, so as to diversify the search if required for problems which require an important amount of iterations for example [Glo97]. The short term memory is characterized by the tabu list, of which the length can be variable. The list functions as a queue in which the move which was added most recently chases the move which was added the longest time ago. The short term aspect is due to the fact that supposing TL^{time} the length of the list, only the TL^{time} previously forbidden moves are kept in memory and are as such set tabu. Both the length of the tabu list and the tabu criterion play an important role in the performance of the tabu search procedure. With regards to the list length, many different options can be considered, however after various tests, solely a few of those have been retained for the present application. The length of the tabu list can be set either statically, dynamically or adaptively. In the first case, the length is simply set to a given number which does not vary with the amount of machines and jobs of the different instances. For the present case, the static value of the tabu list length was chosen to be 32 when using move-tabu and 25 for task-tabu after experimenting with several values. Both move-tabu and task-tabu are to be detailed in the upcoming paragraphs. In the case of a dynamic tabu list length however, the length of the tabu list is set dependently to the amount of of jobs n and machines m. In his tabu search implementation for the JSP, Taillard suggested the tabu list length be set dynamically to $\frac{(n+m)}{2}$ [Tai94]. Another approach is that of adaptive tabu list length, which can adapt during the search. An adaptive approach worthwhile mentioning is that of Dell'Amico and Trubian adapted for the purpose of this work [DT93]. Starting with a length of $\frac{(n+m)}{2}$, the length of the tabu list depends on three different rules. Firstly, if the objective value of the current solution is less and as such better than the best objective value found precedently, the list length is set to 1. Secondly, if the search is currently situated in a so-called improving phase, in which the objective value of the current solution is lesser than in the previous iteration, and if the current list length is greater than a given minimum threshold $TH_{min} = \frac{(n+m)}{2} - \frac{(n+m)}{4}$, then the list length is to be decreased by 1. Inversely, if the search is currently situated in a deteriorating phase, in which the objective value of the current solution is greater than or equal to that of the previous solution, and if the current list length is lesser than a given maximum threshold $TH_{max} = \frac{(n+m)}{2} + \frac{(n+m)}{4}$, then the list length is to be increased by 1. By setting less moves tabu, neighbors close to ones currently providing progressively better solutions are explored and not "overjumped". Conversely, by allowing for more moves to be set tabu, it is hoped that the search can better escape local optima as it will be able to make larger "jumps". As soon as a global optimum is found, the list length is reinitialized by being set to 1. Another adaptive list length method developed for the purpose of this work functions as follows: if after a given amount of 20 iterations no new best solution has been found, then the list length is to be reduced to $\frac{1}{2} \cdot \frac{(n+m)}{2}$ and as soon as a new best solution is found, the list length is set back to its starting length of $\frac{(n+m)}{2}$. The present work implements all of the list length methods mentioned above.

Once the length of the tabu list has been determined, one must determine a move operator and a navigation strategy, respectively defining which neighbors to consider, and which to select. In the present case, the move operator relying on neighborhood structure N_1 detailed in subsection 2.3 is used in the implemented tabu search heuristic. As for the navigation strategy, most tabu search heuristics make use of the steepest-descent method presented in subsection 3.1, that is, for the JSP, they select the allowed neighbor with the smallest makespan. For this reason, steepest-descent is used as navigation strategy in the present tabu search implementation.

Before proceeding to a swap, one must approximate the makespan that would result from the swap and compare this result with the best current neighboring solution as mentioned in 2.4. Once the makespan values have been calculated and the neighbor with minimal makespan has been selected, an addition to the tabu list must be carried out so as to avoid potential return to a local optimum. As such, the criterion according to which moves are set tabu must also be defined. For a swap between operations fo and so, a proposed criteria could be to set the swap move which just occurred and interchanged the processing order of fo and so, as tabu. This criteria shall be referred to as move-tabu. Alternatively, one could forbid not the swap but rather set the second operation involved in the swap as tabu, as such, in the case of consequent critical tasks fo and so, so would be added to the tabu list and the different maximum tabu list length variations. This second criteria is to be called task-tabu for the purpose of the present work. Both criteria in combination with the various tabu list length methods shall be implemented and results be compared in section 5. The tabu search meta-heuristic implemented as part of this work is represented as a flow chart in Figure 11 in the Appendix. A brief summary of the functioning of the tabu search algorithm is provided as follows: first the algorithm starts with an existing solution, it then calculates an approximation of the makespan of solutions which do not make use of moves or tasks present in the tabu list, and in which a pair of critical tasks could be swapped as defined in N_1. If the approximated makespan is better than the best neighbor, proceed with the swap and verify with the exact makespan. Then, steepest-descent is applied, which leads to the neighboring solution with the smallest makespan to be selected as the new solution. Finally, either the move which just occurred is set tabu if move-tabu is used, or the second task part of the move is set tabu if task-tabu is used, hence the tabu list is updated with respect to the tabu list length. This process is repeated until a maximum amount of 50 consecutive iterations without finding any new best solution is reached or all neighboring solutions are set tabu, after which the algorithm returns the solution with the shortest makespan it found. Figure 5 provides a visual comparison of how local search and tabu search compare in their search process, the graph was constructed using a randomly generated instance. The generation of such an instance is detailed in subsection 4.1. The local search procedure, depicted by the gray curve gets stuck at a local optimum as soon as in iteration 5 and terminates its search. The orange curve depicts the optimal makespan, and as can be seen, the tabu search curve (in blue) continues the search where local search stopped, which allows it to iterate through many more solutions, until an end criteron is attained (in this case no neighboring solution which has not been set tabu is available).

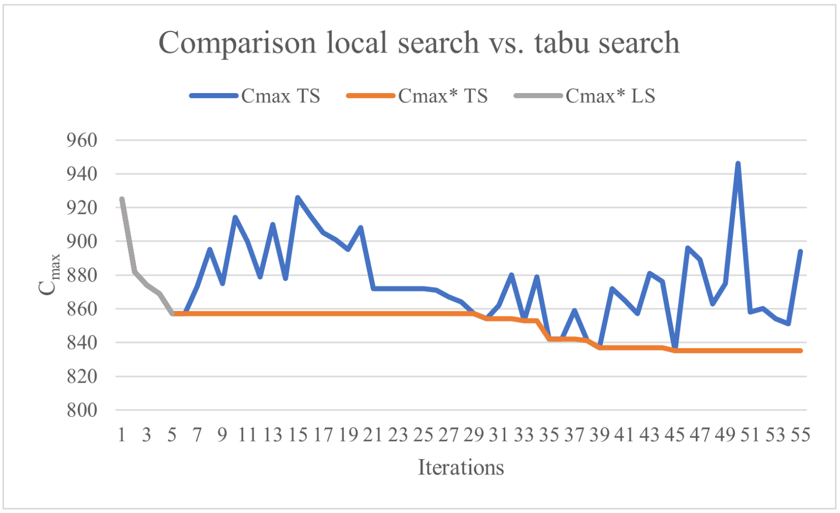


Figure 5: Comparison between local search and tabu search

3.3 Constraint programming: OR-Tools solver

This sections aims at providing a very broad overview of constraint programming and to mention which constraint programming solver is to be used. Constraint programming is among the most successful solving methods for combinatorial optimization problems such as scheduling problems, among which the JSP and several of its extensions, as shown in [LBL$^+$20] and [MMKP22] for instance. The underlying concept of constraint programming is that of allowing the user to state the problem through constraints which must be satisfied. A constraint satisfaction problem (CSP) consists of a set of variables, a set of domains representing the range of values for the variables, and a set of constraints describing specific relations the variables should hold to eachother. A solution to a CSP problem is an assignment of variables to given values, such that the variable's domains and the constraints are respected. As soon as a CSP is refined with an objective function to be optimized, the CSP becomes a constraint optimization problem (COP). The particularity of constraint programming lies in the fact that it can find all optimal solutions of a problem, which is why the present work also implements a constraint programming approach, so as to generate all optimal solutions in order to comparatively evaluate the solutions found through the (meta-)heuristical approaches. Constraint programming relies on two main aspects: constraint propagation and search. Constraint propagation allows to avoid or remove inconsistencies. Among the local consistency conditions, one of the most relevant in practice is the arc consistency. Considering a constraint, a value assigned to a variable for which this constraint states a relation, has a support if there exist values for the other variables for which the constraint states a relation, such that the constraint is satisfied [RVBW08]. As such, a constraint can be called arc consistent if every value of the variables of the constraint has a support. By removing unsupported values from the domains, the search space diminishes. Another constraint propagation method which avoids inconsistencies is forward checking. As for the search dimension of constraint programming, backtracking search is most commonly used. This approach involves a depth-first search. The constraint programming solver used in the present work is the open source solver CP-SAT Solver from Google OR-Tools `https://developers.google.com/optimization/cp/cp_solver`.

4 Method

Methods which have found practical application as part of this work are to be detailed in the present section. Firstly, several instances for the JSP have been generated following the procedure described in subsection 4.1, then, first feasible solutions for these instances have been calculated using Giffler and Thompson's algorithm with varying priority rules in similar fashion as mentioned in subsection 2.1 and shown in Table 2. Then, a local search heuristic has been developed, which corresponds to the steepest-descent navigation strategy using the N_1 neighborhood structure, both respectively detailed in subsections 2.3 and 3.1. Furthermore, a tabu search meta-heuristic has been developed. This tabu search makes use of the precedently introduced local search heuristic and enriches it with a tabu list. Several tabu list length approaches namely static, dynamic and adaptive are implemented, all of which have been mentioned in subsection 3.2. Finally, the constraint programming solver CP-SAT from Google OR-Tools has been used for solving the generated instances in order to obtain state-of-the-art solutions to compare the solutions of the (meta-)heuristics to, after letting these solve the generated instance problems as well. The computing of these results occurred on a laptop with the following specifications: Inter Core i7-10510U @ 1.80 GHz 2.30 GHz, 16 GB ram processor and Windows 10 OS. The programming language was Python.

4.1 Instances

The procedure to generate random benchmark problem instances for the JSP is to be described in the following subsection. It relies on Taillard's work [Tai93], in which the amount of machines m alternates between 15 and 20 and the amount of jobs n is either 15, 20, 30, 50 or 100. The various combinations of m and n result in 8 distinct blocks and for each block 10 instances are generated, which is a particular and deliberately chosen structure relying on Taillard's work. Each task of a job must be processed on a different machine and as such, the amount of tasks for each job is equal to the amount of machines. The processing time pt of a task of a job is defined at random by the means of a unified distribution $U[1, 99]$. Table 3 displays each of the 8 blocks and their corresponding amount of machines m, jobs n, and amount of instances generated.

Block	Nb. Machines	Nb. Jobs	Nb. Instances
1	15	15	10
2	20	15	10
3	20	20	10
4	30	15	10
5	30	20	10
6	50	15	10
7	50	20	10
8	100	20	10

Table 3: Blocks of benchmark instances for the JSP

While back in his time Taillard had to make use of a self implemented linear congruential generator to generate random numbers, this is not the case in the work presented here. The random numbers and hence instances generation occurred with the help of Python's *random* library. Let Inst denote a JSP instance, Shuflist a random machine ID list, job a job instance and op an operation. An overview of the implemented code for the generation of one instance with specific amount of machines and jobs is provided below in Algorithm 5:

Algorithm 5: Generate JSP problem instance

```
Input: m amount machines, n amount jobs
Output: Inst(JSP-Instance)
1  $Inst \leftarrow \{\}$
2  $Shuflist \leftarrow \{machines\}$
3  for $i = 0$ to $n-1$ do
4      $job \leftarrow \{\}$
5      random shuffle from Shuflist
6      forall $j \in Shuflist$ do
7          $pt \leftarrow Randomnumber \in \{1, ..., 99\}$
8          $op \leftarrow (j, pt)$
9          $job \leftarrow job \cup op$
10     $Inst \leftarrow Inst \cup job$
```

In order to ensure that the machine sequences of the orders differ, a shuffle list is created. This consists of all machine numbers and is randomly shuffled after each operation iteration. The machine sequence of an operation then emerges from this list. Each operation is then assigned a machine ID and a processing time between 1 and 99 minutes (see lines 2-8).

4.2 Performance measurement

In order to measure the performance of the various methods applied presently, the best possible makespan found through a method is used, the lower the makespan, the better the solution quality. A method's performance is expressed in percentage of improvement or decline compared to another method. This allows for the quantification of relative performance. The analysis of the performance of different priority rules used in the Giffler and Thompson algorithm can be done firstly through the absolute makespan values and secondly by measuring the mean relative improvement (MRI) for each method, that is, how well the makespan has been reduced after applying tabu search to a starting Giffler and Thompson solution. The comparison between CP-solver against local search against tabu search also makes use of the MRI performance metric. The measurement of the performance of tabu list length methods and tabu criteria also occurred using MRI.

5 Results

This section presents results obtained by applying the several aforementioned (meta-)heuristics and CP-solver to randomly generated JSP instances. The four main aspects which have been put under scrutiny are the following: firstly the different priority rules to be used when applying the Giffler and Thompson algorithm as presented in subsection 2.1, secondly the various parameters which have been varied for the tabu search implementation as described in subsection 3.2, thirdly the impact of priority rules on the efficiency of tabu search, and finally the performance in delivering the smallest possible makespan.

5.1 Priority rules

It is of great importance for the later implementation of local search based heuristical approaches to begin their search with a best possible first feasible solution. For this reason, several priority rules to be used in Giffler and Thompson's algorithm have been investigated for performance by analyzing 10 instances for each of the 8 blocks without any time limitation. This was done by measuring the makespan delivered by the algorithm for the 8 different blocks of instances. Following priority rules have been tested: Shortest Processing Time (SPT), Most Work Remaining (MWKR), Shortest Processing Time for Operations (SPT op), and Random. For each of the blocks, the MWKR rule delivered the lowest makespan among all rules as can be seen in Table 4 and Figure 6. Hence, over the 80 instances, MWKR remained unbeaten. Note that the Giffler and Thompson algorithm is mentioned as GTA in Figure 6 below.

	Priority rules			
Block	SPT	MWKR	SPT_op	Random
1	2067	1504	1954	1750
2	2479	1744	2362	2046
3	2857	2009	2657	2438
4	3137	2277	3066	2623
5	3575	2512	3335	2876
6	4495	3195	4510	3514
7	5016	3566	4925	4019
8	7844	6117	8628	6548

Table 4: Makespan for Giffler and Thompson algorithm with different priority rules

It would therefore appear that selecting the MWKR rule when generating a first feasible solution for the JSP would be the most appropriate option so as to generate the best possible starting solution. Interestingly, the Random rule displayed better results than both variations of the SPT rule. One can only assume the reason the Random rule performed as well is due to it randomly leading to as good decisions as the MWKR rule. Of great interest is to note that the run time lies within less than 1 second and even in the milliseconds range for smaller instances, thus the creation of an admissible solution appears to be very fast.

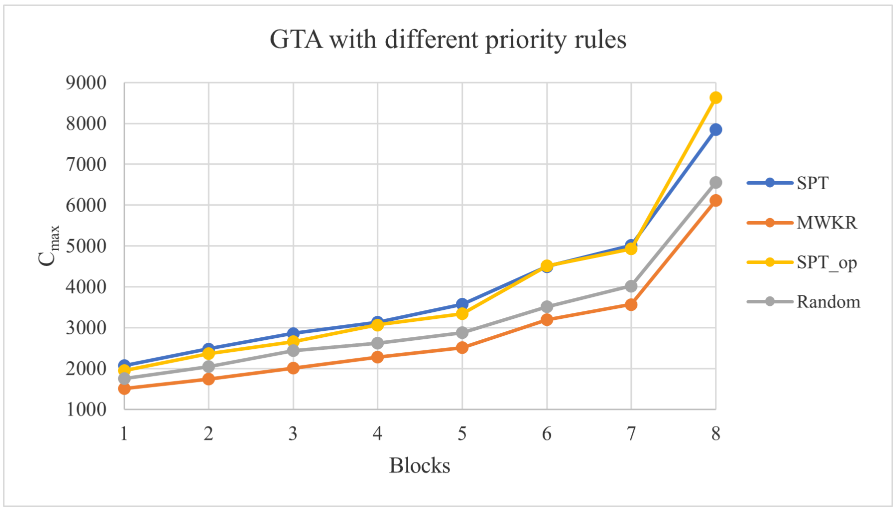


Figure 6: Giffler and Thompson algorithm using different priority rules

5.2 Tabu parameter

So as to compare the performance of the different tabu list approaches, the following procedure has been developed. In a first time, among the 8 blocks, solely the middle ones were kept for analysis, after which 12 instances for each of these blocks were generated. The selected blocks are as follows: block 3 with 20 machines and 20 jobs, block 4 with 30 machines and 15 jobs, and block 6 with 50 machines and 15 jobs. The conclusions from the comparison among the middle blocks shall be applicable to the smaller and larger instances as well. The analysis occured using the MWKR priority rule and a time limitation of 180 seconds. This generous time limitation ensures that the tabu list length can attain a sufficient length for the different parameters which are to be analyzed, to have an influence on. Table 5 below shows the MRI between several implementations of tabu search with various list length methods and tabu criteria compared to solutions obtained by applying the Giffler and Thompson algorithm with MWKR priority rule. Note that through the following analyses, tables and graphs, the run time is always provided in seconds. The move-tabu criteron is found abbreviated in Table 5 as M-T and task-tabu as T-T.

		Static		**Dynamic Taillard**		**Adaptive Dell'Amico**		**Adaptive own**	
	Block	Run time	MRI	Run time	MRI	Run time	MRI	Run time	MRI
M-T	20,20	27.5	11.74%	29.8	11.83%	26.9	12.21%	30.3	12.16%
	30,15	32.1	7.53%	42.0	8.70%	38.0	8.57%	24.8	6.86%
	50,15	53.2	3.80%	52.3	3.80%	53.7	3.80%	47.7	3.62%
	Ø	37.6	7.69%	41.4	8.11%	39.5	8.19%	34.3	7.55%
T-T	20,20	20.0	11.06%	20.9	11.20%	23.9	11.49%	35.0	12.82%
	30,15	37.4	9.63%	30.8	9.48%	38.6	9.71%	38.4	10.05%
	50,15	54.6	3.96%	83.1	4.53%	82.3	4.97%	45.0	3.57%
	Ø	37.3	8.22%	44.9	8.40%	48.3	8.72%	39.5	8.81%

Table 5: Tabu list length methods for different tabu criteria

The MRI values differ from one another within a range of less than 2%. These are rather small differences. This does not necessarily imply that these differences are unsignificant as 2% can translate into great absolute values in practical applications in factories for example. When comparing move-tabu and task-tabu, it appears that the latter provides better improvements than the former on average, although here as well, the differences lie within the 2% range mentioned above for most cases, with a single exception for block (30,15) and the own adaptive method. Overall, the results show that the static list length methods perform worse in terms of improvement than the dynamic methods which in turn achieve a lesser improvement than the adaptive methods. This conclusion was to be expected as the dynamic and adaptive methods are of greater complexity and thus, can adjust themselves to the different instances in better fashion than a static list length method. The combination of task-tabu and the self developed adaptive list length method provided the better MRI values across all combinations, while having similar running time as other options. It appears however, that the own adaptive method struggles when dealing with greater instances such as those in block (50,15). In this specific case, the best MRI value is achieved by the combination of task-tabu with Dell'Amico's adaptive method. The conclusions to be drawn are as follows: for instances with amount of jobs $n < 50$, tabu criteria task-tabu in combination with self developed adaptive list length method are to be applied, for instances with amount of jobs $n \geq 50$, task-tabu with Dell'Amico's adaptive list length method should be preferred.

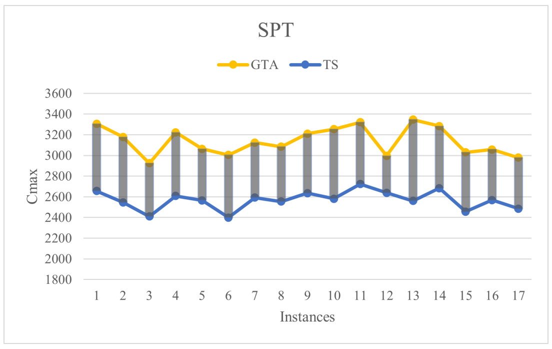
5.3 Impact of priority rules on tabu search

An interesting analysis is that of the comparison of quality improvement for the different starting solutions with tabu search. The analysis occurred based on 17 instances from block (30,15) with no time limitation. A first finding is that applying tabu search by starting with a solution generated using the Giffler and Thompson algorithm with the MWKR priority rule, will provide the least improvement as shown in Table 6.

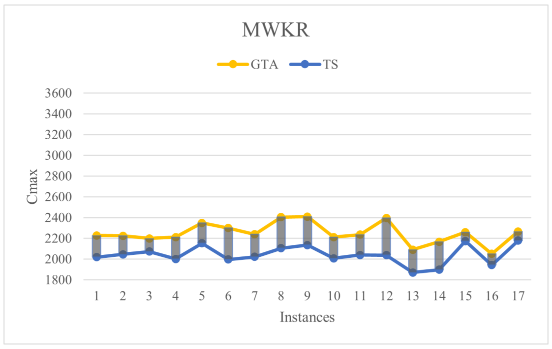
	SPT	MWKR	SPT_op	Random
MRI	18%	9%	21%	16%

Table 6: Mean relative improvement of makespan using different priority rules

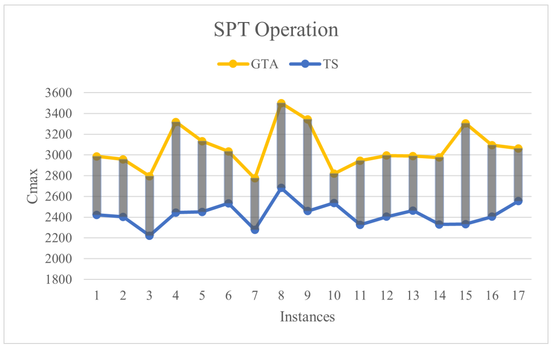
The quality improvement for the different starting solutions which depend on the priority rule used, after implementing tabu search, is shown in further detail in Figure 7. Note that TS refers to to tabu search. As the MRI values are different among priority rules, it would appear that the improvement provided through tabu search is dependent on the quality of the starting solution. The better the starting solution, the lesser the improvement through tabu search. Conversely, the worse the starting solution, the better the improvement through tabu search. As for the starting solutions, these were also systematically of greater quality when using the MWKR rule than for any other rule. The makespan values after tabu search are also best on average when MWKR was used to generate the starting solution. When comparing Subfigure 7a and Subfigure 7b, one can observe that even the starting solutions generated using MWKR are of better quality than those generated using SPT and improved through tabu search. It is also visually noticeable that the distance between the makespan provided by the Giffler and Thompson algorithm using MWKR and the makespan provided by using tabu search is the least important among all instances, as compared to other rules. As can also been seen when comparing Subfigure 7b with Subfigures 7a, 7c and 7d, the distance from the yellow curve (starting solution) to the blue curve (solution improved through tabu search) is considerably greater than for the MWKR rule for all the other rules. Furthermore, there exists 3 out of 17 instances: 3, 12 and 15, for which the Random rule provided a starting solution which, after tabu search, provided better results than when starting with the MWKR solution as seen in comparing Subfigures 7b 7d. Neither SPT nor SPT op achieve a such performance in any instance as seen in Subfigures 7a and 7c. When comparing the 17 instances between the SPT and SPT op rules, it occurred five times that a worse starting solution achieved a better makespan. The conclusion of the present analysis is as follows: the MWKR rule is to be preferred in most cases and the better the starting solution, the better the makespan delivered through tabu search.



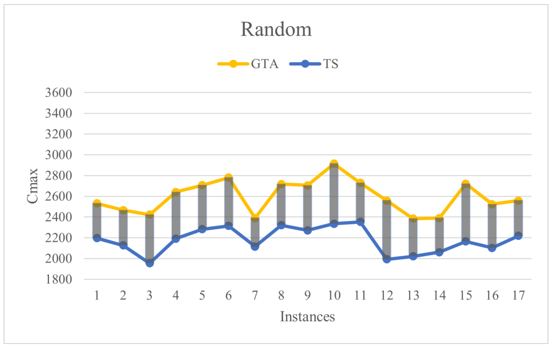
(a) Shortest processing time rule



(b) Most work remaining rule



(c) Shortest processing time of operation rule



(d) Random rule

Figure 7: Comparison of makespan of Giffler and Thompson against Tabu search for different priority rules

5.4 Performance comparison

The performance in terms of makespan are to be detailed in the following subsection. The analysis was performed on 8 different blocks with 10 instances each and a time limitation of 45 seconds for the CP-solver and the tabu search. The local search procedure was constrained by no time limitation as it always terminated before the 45 seconds constraint. Table 7 below details the average run time in seconds and average makespan achieved by the CP-solver (mentioned as CP in the table) across the different blocks. Then, the relative solution quality of the makespan found using the Giffler and Thompson algorithm, local search (as LS), and tabu search is provided in percentage of difference to the other method's solutions. In that sense, the 21.3% value in the first line states that the makespan found using local search is on average 21.3% greater than the one found by using the CP-solver for the instances of block 1. Within the 45 seconds time limit, the CP-solver manages to solve the problem instances optimally 13 out of 80 times, mostly within the smaller blocks and more rarely within the middle blocks. All methods require greater run time to solve the greater instances and even the CP-solver does not find an optimal solution within 30 minutes run time for larger instances. The greater run time of tabu search for larger instances is due to the fact that a larger amount of swaps is required since the amount of operations is more important. As expected, the Giffler and Thompson algorithm delivered worse makespans than local search, which in turn provided worse makespans than tabu search. Perhaps more interesting is the fact that the Giffler and Thompson algorithm requires less than a second to determine a starting solution which requires on average 2.3 seconds to be improved by 3.3% through local search, and 34 seconds to be improved by 7.8% through tabu search. Noteworthy is also that the improvement of the solution through tabu search compared to local search amounts to 4.6%, while tabu search requires 32 seconds more than local search to run, which represents 17 times the run time of local search.

	CP-solver		**GTA**	**LS**			**TS**			
Block	Run time	Cmax	% CP	Run time	% CP	% GTA	Run time	% CP	% LS	% GTA
1	24	1175	28.0%	0.4	21.3%	-5.2%	10	13.9%	-6.1%	-10.9%
2	44	1393	25.2%	0.5	20.3%	-3.9%	21	12.1%	-6.8%	-10.5%
3	45	1609	25.0%	1.3	18.1%	-5.4%	32	9.6%	-7.1%	-12.2%
4	35	1866	22.2%	1.2	17.9%	-3.5%	34	10.4%	-6.3%	-9.6%
5	45	2061	21.9%	2.4	17.1%	-4.0%	44	10.9%	-5.2%	-9.0%
6	44	2882	10.9%	2.7	9.0%	-1.7%	42	6.3%	-2.4%	-4.1%
7	45	3239	10.2%	3.3	7.7%	-2.3%	45	5.0%	-2.5%	-4.7%
8	45	6080	0.7%	6.3	-0.2%	-0.8%	45	-0.5%	-0.3%	-1.1%
Ø	41	2538	18.0%	2.3	13.9%	-3.3%	34	8.5%	-4.6%	-7.8%

Table 7: Relative solution quality comparison between GTA, LS, TS and CP-solver

As depicted in Figure 8, the CP-solver systematically provides better solutions than the other methods except for block 8. In exactly 6 instances out of 80, all of which within block 8, tabu search manages to deliver a smaller makespan than the CP-solver. As for the 13 instances for which the CP-solver found the optimal solution, tabu search finds a solution on average 9.5% greater than that of the CP-solver. As a general rule, it appears however that the greater the instances, the lesser the difference between CP-solver solution and tabu search solution, displaying a decrease from 13.9% in the first block to 5% in the seventh block, and the last block being that in which tabu search even performs (minimally) better than the CP-solver. As such, providing a better makespan than the CP-solver through metaheuristical approach appears to be a great challenge for the smaller instances, while greater instances seem to provide more potential. As can be observed for block 3, with $n = 20$ and $m = 20$, the CP-solver performance is not as significantly better as for the previous and following blocks. The struggles of the CP-solver with the instances of block 3 can be observed in the form of a dip in Figure 8. This might be due to the fact that $n = m$, however this hypothesis deserves further investigation. To put the results of the present analysis into perspective, one could be tempted to mention that the implemented tabu search can still be refined in some aspects and that the CP-solver delivers significantly good solutions within very short time.

The conclusion of the present analysis can be stated as follows: if the goal is to obtain a robust solution within a very short period of time, then the Giffler and Thompson algorithm is of great relevance. If the period of time can be increased by a few seconds, applying local search subsequently can be of great interest. If a better solution with a somewhat more running time is desired, then tabu search shoud be used. The CP-solver is to be used if optimal and similarly good solutions are desired, but should be considered with caution for very large instances.

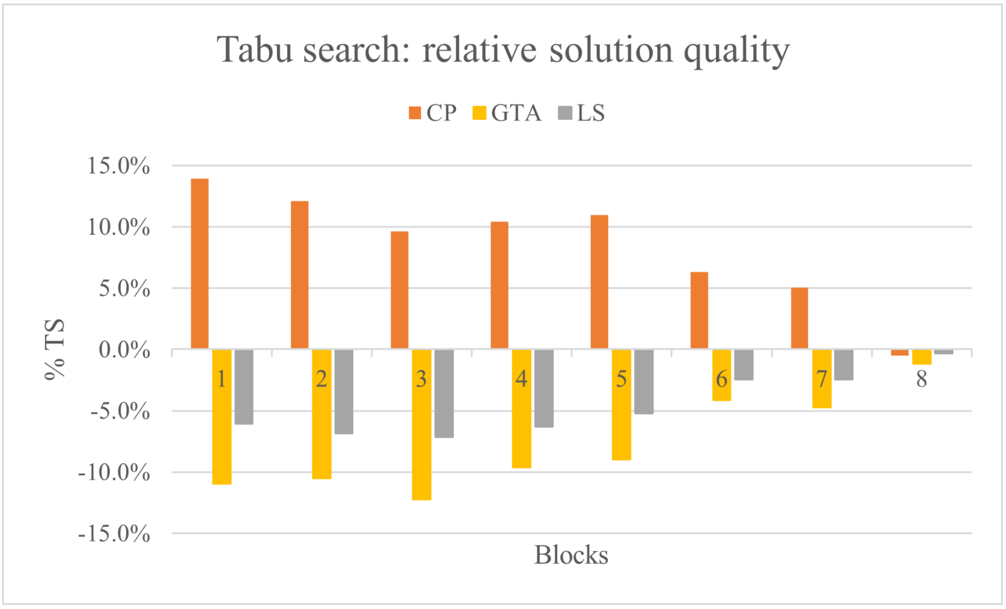


Figure 8: Quality of tabu search solutions in relation to other methods

6 Limitations and future investigations

The aforementioned results are to be interpreted with caution for several reasons, which the present section will aim at briefly explaining. Firstly, the analyses performed as part of the present work and their subsequent results rely on the specific structure of the generated benchmark instances. As such, other results relying on different types of instances are not to be compared with the ones provided above. Furthermore, the limitation of running time for the different methods used to solve the JSP that are local search, tabu search and CP-solver is also to be considered. If one decides to set a different time limit, different results may arise. For comparison purposes, if the limitation of running time was to be set to 7 seconds instead of 45, then the tabu search procedure would provide a better makespan than the CP-solver in 19 out of 80 instances (compared to 6 for the 45 seconds limit), all of which are situated starting from block 5. As such, depending on the application purpose of the presented methods, the tabu search meta-heuristic might very well be a fair alternative to the CP-solver. It should be remarked that several additional measures could be taken in order to increase the performance of the tabu search meta-heuristic: from more precise parameter tuning to adjustments in the data structuration, as well as elements pertaining to the implementation of variable neighborhood structures during the search for instance. Furthermore, it could be of great interest to implement different types of meta-heuristics such as simulated annealing or ant colony algorithms for example, so as to compare their performance not only to the CP-solver, but also to the tabu search approach. Of major interest might as well be to apply the presently introduced methods to different variations of the JSP as for example one which takes setup times into account. Tabu search might perhaps have better chances at beating the CP-solver under such circumstances.

References

[ABZ88] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.

[AvLLU94] Emile HL Aarts, Peter JM van Laarhoven, Jan Karel Lenstra, and Nico LJ Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[BDP96] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research*, 93(1):1–33, 1996.

[BEP$^+$19a] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and J Weglarz. Flow shop scheduling. In *Handbook on scheduling*, pages 271–320. Springer, 2019.

[BEP$^+$19b] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and J Weglarz. Open shop scheduling. In *Handbook on scheduling*, pages 321–341. Springer, 2019.

[BEP$^+$19c] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and J Weglarz. Scheduling in job shops. In *Handbook on scheduling*, pages 345–401. Springer, 2019.

[BJS94] Peter Brucker, Bernd Jurisch, and Bernd Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics*, 49(1-3):107–127, 1994.

[CK16] Imran Ali Chaudhry and Abid Ali Khan. A research survey: review of flexible job shop scheduling techniques. *International Transactions in Operational Research*, 23(3):551–591, 2016.

[DT93] Mauro Dell'Amico and Marco Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations research*, 41(3):231–252, 1993.

[Glo86] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[Glo97] Fred Glover. Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in computer science and operations research*, pages 1–75, 1997.

[GT60] Bernard Giffler and Gerald Luther Thompson. Algorithms for solving production-scheduling problems. *Operations research*, 8(4):487–503, 1960.

[Hau89] Reinhard Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, 1989.

[JP19] Florian Jaehn and Erwin Pesch. Job shops. In *Ablaufplanung: Einführung in Scheduling*, pages 98–116. Springer, 2019.

[KP19] Georgios M Kopanos and Luis Puigjaner. Planning and scheduling. In *Solving large-scale production scheduling and planning in the process industries*, pages 3–8. Springer, 2019.

[LBL$^+$20] Willian T Lunardi, Ernesto G Birgin, Philippe Laborie, Débora P Ronconi, and Holger Voos. Mixed integer linear programming and constraint programming models for the online printing shop scheduling problem. *Computers & Operations Research*, 123:105020, 2020.

[Lew83] Harry R Lewis. Michael r. πgarey and david s. johnson. computers and intractability. a guide to the theory of np-completeness. wh freeman and company, san francisco1979, x+ 338 pp. *The Journal of Symbolic Logic*, 48(2):498–500, 1983.

[LGPI97] Shyh-Chang Lin, Erik D Goodman, and William F Punch III. Investigating parallel genetic algorithms on job shop scheduling problems. *Lecture Notes in Computer Science*, 1213:383–394, 1997.

[LH21] Dung-Ying Lin and Tzu-Yun Huang. A hybrid metaheuristic for the unrelated parallel machine scheduling problem. *Mathematics*, 9(7):768, 2021.

[MMKP22] David Müller, Marcus G Müller, Dominik Kress, and Erwin Pesch. An algorithm selection approach for the flexible job shop scheduling problem: Choosing constraint programming solvers through machine learning. *European Journal of Operational Research*, 2022.

[PW06] Yves Pochet and Laurence A Wolsey. *Production planning by mixed integer programming.* Springer Science & Business Media, 2006.

[RRZ$^+$19] Ravi Ramya, Chandrasekharan Rajendran, Hans Ziegler, Sanjay Mohapatra, K Ganesh, et al. Capacitated lot sizing problems in process industries. In *Algorithm Engineering*, pages 34–42. Springer, 2019.

[RVBW08] Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

[SGV12] Veronique Sels, Nele Gheysen, and Mario Vanhoucke. A comparison of priority rules for the job shop scheduling problem under different flow time-and tardiness-related objective functions. *International Journal of Production Research*, 50(15):4255–4270, 2012.

[Tai93] Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285, 1993.

[Tai94] Eric D Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA journal on Computing*, 6(2):108–117, 1994.

[UZ12] Thomas Uher and Adam Zantis. The critical path method. In *Programming and scheduling techniques*, pages 36–58. Routledge, 2012.

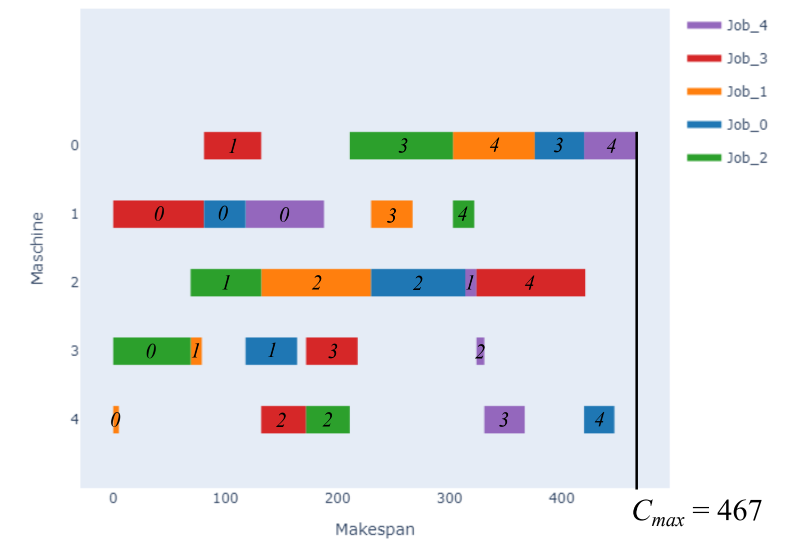
[Wat03] Jean-Paul Watson. Local search and the job-shop scheduling problem. In *Empirical modeling and analysis of local search algorithms for the job-shop scheduling problem*, pages 19–35. Colorado State University, 2003.

[YN96] Takeshi Yamada and Ryohei Nakano. Job-shop scheduling by simulated annealing combined with deterministic local search. In *Meta-Heuristics*, pages 237–248. Springer, 1996.

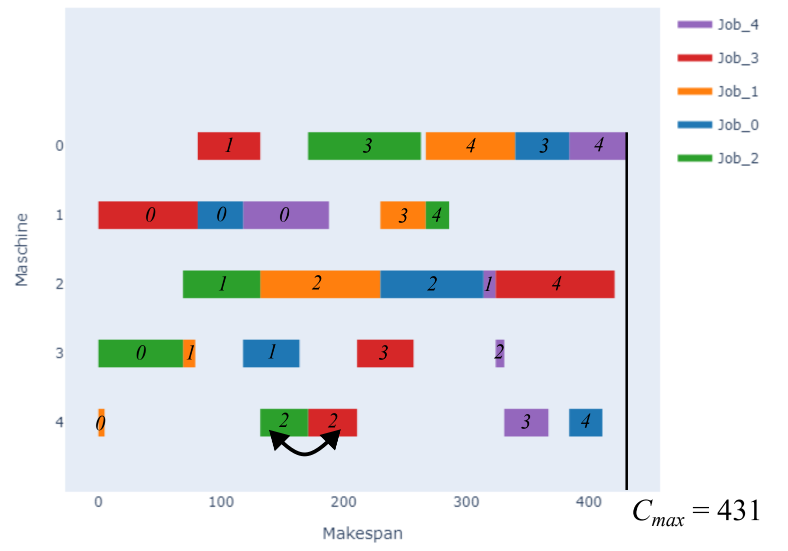
[YN97] Takeshi Yamada and Ryohei Nakano. Job shop scheduling. *IEE control Engineering series*, pages 134–134, 1997.

[ZDZ$^+$19] Jian Zhang, Guofu Ding, Yisheng Zou, Shengfeng Qin, and Jianlin Fu. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing*, 30(4):1809–1830, 2019.

7 Appendix



(a) Gantt chart with $C_{max} = 467$



(b) Gantt chart with $C_{max} = 431$

Figure 9: Example application of neighborhood definition N_1

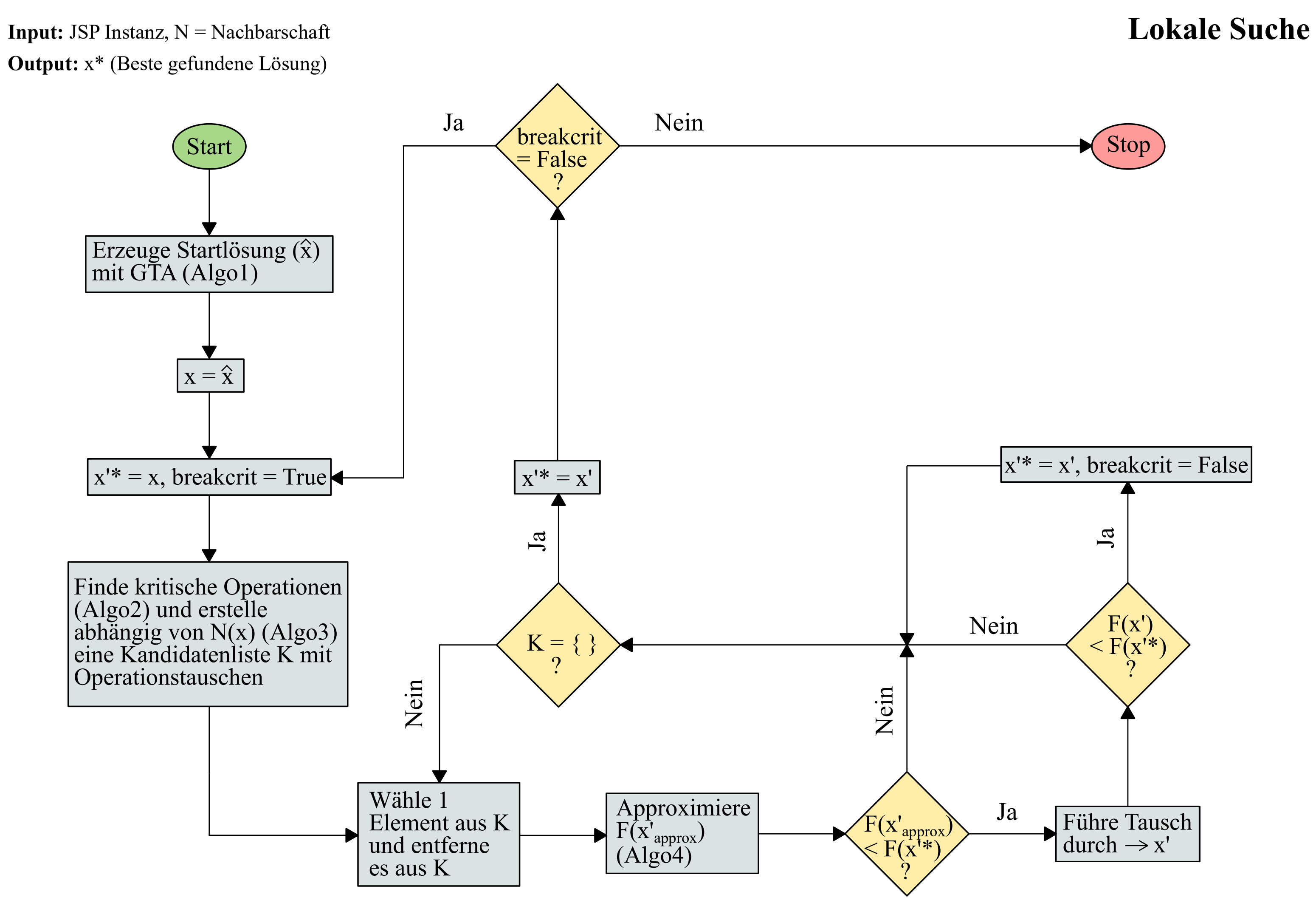

Lokale Suche
Input: JSP Instanz, N = Nachbarschaft
Output: x* (Beste gefundene Lösung)
Start
Erzeuge Startlösung ($\hat{x}$) mit GTA (Algo1)
$x = \hat{x}$
x'* = x, breakcrit = True
Finde kritische Operationen (Algo2) und erstelle abhängig von N(x) (Algo3) eine Kandidatenliste K mit Operationstauschen
Wähle 1 Element aus K und entferne es aus K
Approximiere F(x'_{approx}) (Algo4)
F(x'_{approx}) < F(x'*) ?
Ja
Nein
Führe Tausch durch → x'
F(x') < F(x'*) ?
Ja
Nein
x'* = x', breakcrit = False
K = { } ?
Ja
Nein
x'* = x'
breakcrit = False ?
Ja
Nein
Stop

Figure 10: Flow chart of Local search algorithm

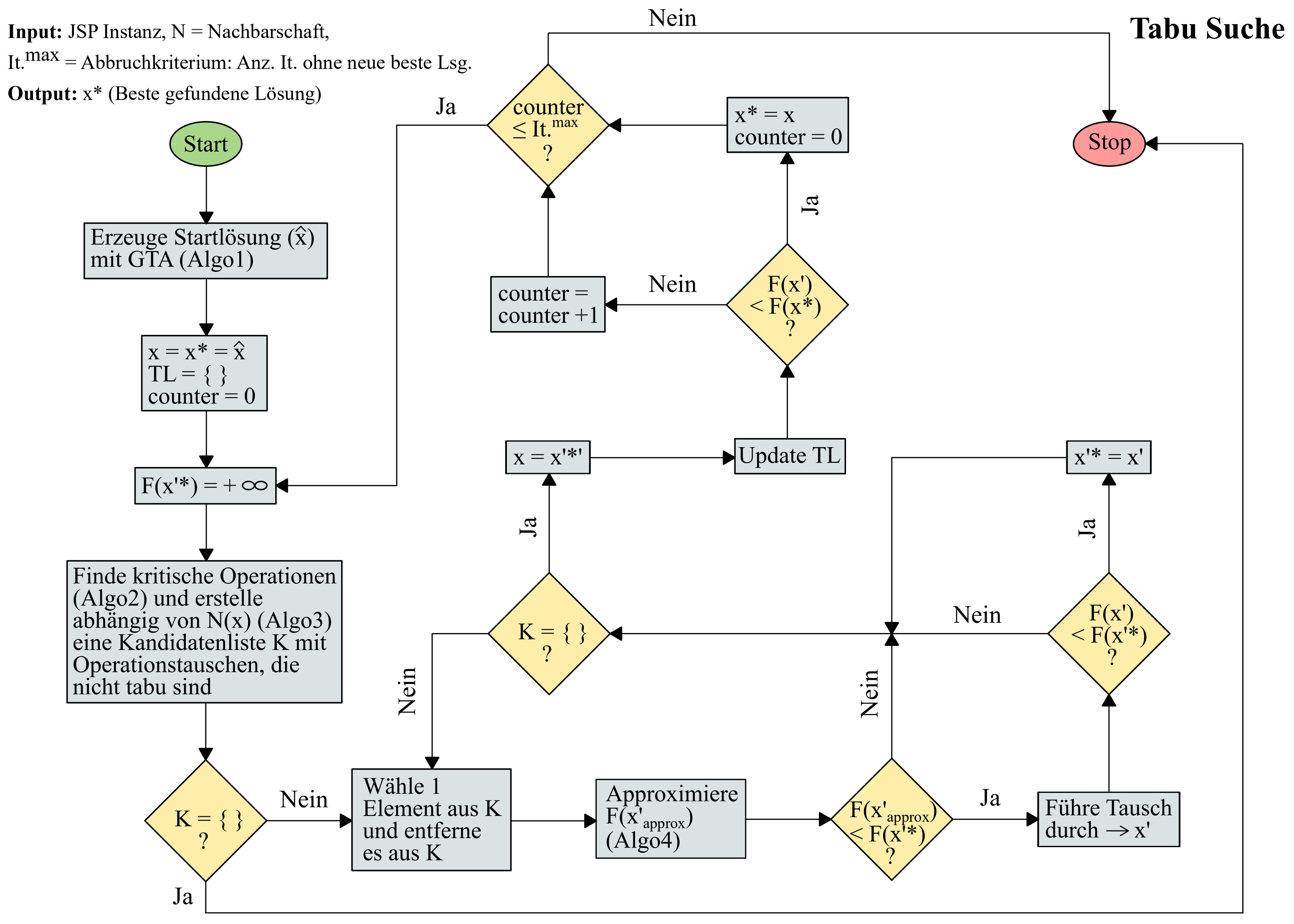

Tabu Suche
Input: JSP Instanz, N = Nachbarschaft,
It.max = Abbruchkriterium: Anz. It. ohne neue beste Lsg.
Output: x* (Beste gefundene Lösung)
Start
Erzeuge Startlösung (x̂) mit GTA (Algo1)
x = x* = x̂
TL = { }
counter = 0
F(x'*) = +∞
Finde kritische Operationen (Algo2) und erstelle abhängig von N(x) (Algo3) eine Kandidatenliste K mit Operationstauschen, die nicht tabu sind
K = { } ?
Ja
Nein
Wähle 1 Element aus K und entferne es aus K
Approximiere F(x'approx) (Algo4)
F(x'approx) < F(x'*) ?
Ja
Nein
Führe Tausch durch → x'
F(x') < F(x'*) ?
Ja
Nein
x'* = x'
K = { } ?
Ja
Nein
x = x'*'
Update TL
F(x') < F(x*) ?
Ja
Nein
x* = x
counter = 0
counter = counter +1
counter ≤ It.max ?
Ja
Nein
Stop

Figure 11: Flow chart of Tabu search algorithm