Analysis of Taboo Search in JSSP

Anders Bjerg Pedersen (andersbp@me.com)

May 29, 2009

Abstract: We present a thorough and broad analysis and review of taboo search in the context of the job-shop scheduling problem. After a quick recap of notation and complexity we compare different solutions techniques and lay out the primary components of taboo search and neighbourhood definitions, and analyse the impact of different solution space factors on running time. Finally we take an in-depth look at the prominent TSAB and i-TSAB algorithms by Nowicki and Smutnicki, analysing the reasons for their efficiency and finally outlining possible future research directions. Throughout the paper we rely heavily on existing literature, giving the contents a survey-like approach.

Introduction

The job-shop scheduling problem (JSSP) is a notoriously difficult problem in combinatorial optimisation. Because of its widespread applicability, however, it has spun huge interest in the research community, from the late 1970's up until today. Its applications include scheduling of jobs in factories (most efficient use of various machines, finishing workloads on time or at the earliest etc.), scheduling employee work hours and many more.

In this paper we shall take a closer look at the problem itself and go through various ways of solving it, especially focusing on the metaheuristic approach taboo search. We examine previous and current solution methods, compare them to each other, point out relevant properties of the JSSP solution space, and briefly suggest ways for further research to follow. Taboo search will play a primal role throughout the paper, as it has proven itself to be extremely efficient in solving JSSP instances compared to other solution methods.

1 Introduction to JSSP

In this section we shall give a brief definition of JSSP, its representation, and its complexity. Throughout this paper we shall maintain the notation given in [9]. A more thorough review as well as examples and overview of notation can also be found in [5].

1.1 Problem definition

JSSP is a generalisation of the flow shop problem, in which we are given a set of n jobs $J = \{1, \ldots, n\}$ and m machines $M = \{1, \ldots, m\}$ on which the jobs are to be processed. Each job has a fixed number of operations, taken from a general set of operations $O = \{1, \ldots, o\}$. Let $l_j = \sum_{i=1}^{j} o_i$ denote the number of operations in jobs 1 through j. In JSSP each operation $i \in O$ has to be processed on a certain machine $\mu_i \in M$ without interrupting the process. We are, however, allowed to pause the jobs in between its operations.

Each operation $i \in O$ has a processing time of $\tau_i > 0$ on machine μ_i. Any given machine can only process one operation at a time, and the order of operations in a job cannot be interchanged (i.e. operation $i + 1$ in a job must be processed completely after finishing operation i). For simplicity we assume no set-up or transfer costs when changing processing of a job from one machine to another.

The general goal of the problem is then to find the minimal makespan of the jobs, i.e. find the processing order that minimises the time needed to process all operations in all jobs. This is also referred to as JSSP with the "makespan criterion".

We can decompose the set of operations O into m subsets, each containing exactly those operations that are to be processed on machine $k \in M$:

$$O = \bigcup_{k \in M} M_k, \quad M_k = \{i \in O \mid \mu_i = k\}, \quad m_k = |M_k|.$$

Denote then the order of the m_k operations on machine k as some permutation of the operations:

$$\pi_k = (\pi_k(1), \ldots, \pi_k(m_k)), \quad k \in M,\ \pi_k(i) \in O.$$

If we let Π_k denote all of these possible permutations of operations on machine $k \in M$, we see that a complete processing order π will be an element of the product set Π of permutations on all machines:

$$\pi \in \Pi = \Pi_1 \times \Pi_2 \times \cdots \times \Pi_m.$$

Representation of the problem can be done most practically by using a directed graph $G(\pi) = (O, R \cup E(\pi))$, as first introduced in [11]. Using the notation from [9], the digraph consists of two types of edges. Edges from the set R (conjunctive edges) represent processing orders of operations in jobs:

$$R = \bigcup_{j=1}^{n} \bigcup_{i=1}^{o_j - 1} \{(l_{j-1} + i, l_{j-1} + i + 1)\}$$

Edges from the set $E(\pi)$ (disjunctive edges) represent processing orders of operations on machines:

$$E(\pi) = \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k - 1} \{(\pi_k(i), \pi_k(i+1))\}$$

The nodes of $G(\pi)$ are the operations from the set O and each node $i \in O$ has weight

τ_i, whereas the edges all have weight zero.

From this representation we have that any graph $G(\pi)$ represents a feasible processing order π only if it contains no cycles, and the corresponding makespan $C_{max}(\pi)$ is equal to the – not necessarily unique – longest path that can be found in the graph. We shall refer to a longest path as the *critical path* of $G(\pi)$.

Example 1: *As a simple example consider a 3-machine, 3-job instance of JSSP with a total of 9 operations:* $J = \{1, 2, 3\}$, $M = \{1, 2, 3\}$, $O = \{1, 2. \ldots, 9\}$. *Job 1 consists of operations 1-4-9, job 2 of operations 5-6-8, and job 3 of operations 2-3-7 (to be processed in that order).*

Machine assignments are as follows: $\mu_1 = \mu_2 = \mu_5 = 1$, $\mu_3 = \mu_4 = \mu_6 = 2$, $\mu_7 = \mu_8 = \mu_9 = 3$. *Processing times are:* $\tau_1 = \tau_4 = \tau_6 = \tau_7 = 2$, $\tau_2 = \tau_3 = \tau_5 = \tau_8 = \tau_9 = 1$.

Below is shown the disjunctive graph representation $G(\pi) = (O, R \cup E(\pi))$ *of a feasible solution* π *to the instance. Notice that the graph is cycle-free.*

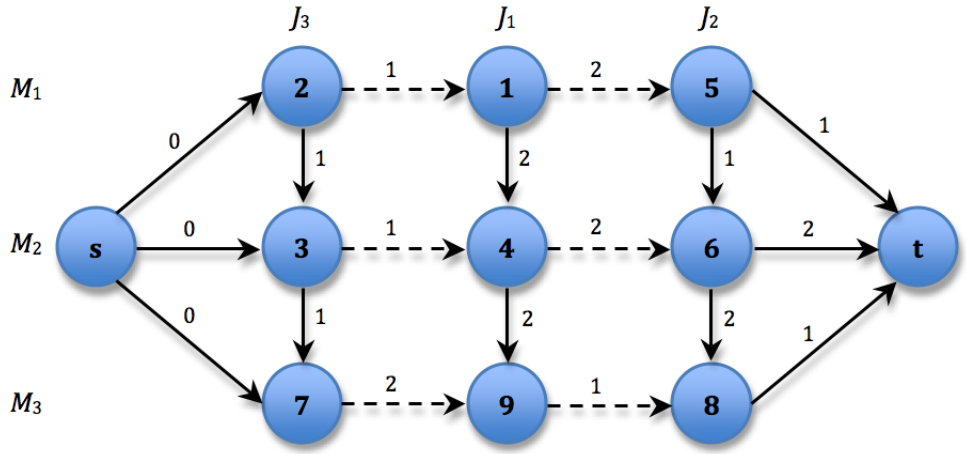

Figure 1: $G(\pi)$ belonging to the instance described. Normal arrows represent edges from the set R, dashed arrows represent edges from the set $E(\pi)$. A source and sink have been added to ease presentation. Also, edge weights are equivalent to the weight of the node from which the edge originates. This eases graphical representation and is equivalent to using node weights in terms of finding critical paths.

From the representation given above, we have the two edge sets:

$$\begin{aligned} R &= \{(2,3),(3,7),(1,4),(4,9),(5,6),(6,8)\} \\ E(\pi) &= \{(2,1),(1,5),(3,4),(4,6),(7,9),(9,8)\} \end{aligned}$$

Notice also in Figure 1 that the single critical path consist of the path $s \to 2 \to 1 \to 4 \to 6 \to 8 \to t$ *of length* $C_{max}(\pi) = 8$.

⊙

We shall return to this representation when discussing blocks and neighbourhoods in sections 3.2 and 4. The impatient reader can also have a look in [5], where another example of this representation is given.

1.2 Complexity

As mentioned before, JSSP represents one of the hardest problems in combinatorial optimisation. In fact, in [6] the authors prove the problem to be NP-complete whenever $m \geq 2$ by reducing it from the 3-partition problem, which is known to be NP-complete. They even generalise their results to show that NP-completeness of JSSP is independent of how one measures the "length" of the problem (either by size of the input or by sum of processing times). Empirical results have confirmed this observation: a fairly simple 10x10 instance (10 jobs, 10 machines) by Fisher and Thompson dating back to 1963 has since shown itself to be notoriously difficult and withstood all attempts to be solved to optimality for 27 years!

There are, however, a few specific instances of JSSP that are in fact solvable in sub-exponential time: ([3], p3)

- Scheduling two jobs by a graphical method.[1]
- The flow-shop problem with $m = 2$ and the additional requirement that every job has the same sequence of machines on which to be processed.
- JSSP with $m = 2$ and each job consists of at most two operations.
- JSSP with $m = 2$ and unit processing times.
- JSSP with $m = 2$ and a fixed number of jobs.

Changing even a slight parameter in the above cases causes the problem to become NP-hard. As soon as the problem grows beyond two machines or outside of the above – as happens in most cases in real life scheduling – the problem can no longer be solved in polynomial time, and therefore it is of much interest in the community to find more efficient solution techniques to JSSP than the ones currently available.

2 Conventional solution techniques

In this section we present a selection of conventional techniques for solving the job-shop problem. We consider both exact methods and heuristics, although the main focus will lie on the latter, where we shall also find taboo search. We base our review primarily on [3] for descriptions and [1] for results and comparisons.

2.1 Branch and bound

Branch and bound is a well-known exact solution method applied to a variety of different optimisation problems. Its main focus lies in being able to enumerate the entire feasible solution space of a problem but quickly cutting away uninteresting sub-solution spaces not yielding better results than previously found solutions. It identifies different solution spaces by recognising certain attributes common to the feasible solutions in that space.

[1] As can be seen in [4], the author reduces the problem to an unrestricted shortest-path problem in the plane with rectangular obstacles. After constructing a special network, it is possible to solve the instance in $O(n \log n)$ time. Here the n obstacles are representations of the operations on machines, and the dimensions of the obstacles represent their processing times.

In the case of the Travelling Salesman Problem, for instance, different solution spaces may be characterised by which edges are allowed in the closed Hamiltonian path in the graph corresponding to the problem instance.

Branch and bound uses estimations in the form of lower and upper bounds on the problem to be able to cut away certain solution spaces. The better bounds, the faster a solution can be found. An initial heuristic is also used to infer an upper bound (in minimisation problems) that can be compared with a lower bound during execution: if the lower bound of a certain solution space exceeds the initial upper bound found by the heuristic, the solution space is of no interest and can be cut away. In the course of the branch and bound algorithm the best bounds found are updated.

The nuts and bolts of branch and bound lie in determining two things: good lower bounds with which the algorithm can cut away branches, and a good and efficient branching strategy; i.e. a way to characterise the different solution spaces from each other. Firstly looking at lower bounds, a large number of proposals exist for JSSP. The different lower bounds are often specialised for different types of JSSP instances, though, and going through them here would be outside the scope of this paper. We mention, however, that attempts to find good lower bounds can be based on various methods, including Lagrangian relaxation, optimal solutions to simpler subproblems (e.g. with fewer machines) and the classic head-tail bounds (see [3], p7f for more details).

In determining a good branching strategy there are again a vast amount of suggestions that have appeared over the course of the last 30 or 40 years. We shall not go deep into these different strategies but only mention that there also exists a branching strategy which is very similar to the one imposed by Nowicki and Smutnicki's taboo search procedure discussed later on. This strategy is based on critical paths and the blocks of which this path consists. When at some sub-problem we need to branch, we do so by moving an operation from a block to either the beginning or the end of that same block.

2.2 Priority Dispatching Rules

Priority dispatching rules, or also known simply as priority rules, are one of the simplest and most frequently used heuristics for JSSP. This heuristic generates feasible solutions by continuously selecting unscheduled operations one at a time based on certain rules. These rules vary from problem to problem and can often be adjusted to match certain instances better than others. Possible rules include operations with shortest remaining job processing times, the first one waiting in the queue and others. An example of such rules can be found in [3], p12.

The algorithm maintains a list $Q(t)$ of operations yet to be scheduled as well as a "conflict set" containing for instance the operations competing for the same machine at the same time and uses priority rules to select the next operation to be scheduled. The operation i to be scheduled is chosen from $Q(t)$ so that it does not conflict with operation j, which is the operation chosen with regards to the priority rule in mention.

2.3 Shifting Bottleneck Procedures

A shifting bottleneck procedure repeatedly schedules operations on individual machines, not immediately taking notice of the overall schedule for all machines. The remaining unscheduled machine with the highest makespan (the bottleneck machine) is scheduled

first. This type of heuristic again makes use of the head and tail values for each operation in order to calculate the makespan of a certain machine. After each machine has been scheduled, the procedure applies a local optimisation procedure to re-optimise the machines that have already been scheduled, before continuing with the next bottleneck machine. The great advantage of this heuristic is the simplicity and speed of its subroutines: solving a one-machine problem can be done efficiently by a branch and bound algorithm.

The efficiency of shifting bottleneck procedures relies heavily on the order in which the individual machines are scheduled. By defining bottleneck as the machine with the longest makespan, one settles for an order of machines not necessarily representing the final optimal schedule. Another slightly modified approach involves using a form of branch and bound to enumerate and schedule the different inclusions of new machines until a certain depth of the search tree. After this depth has been reached, several machines beside the bottleneck one are included in every branch, hereby making the number of branches substantially smaller. The reason for this extended rule is of course that enumeration of all possible inclusions of machines is computationally infeasible.

2.4 Local Search

Local search procedures is a generic term for such methods as hill-climbing, simulated annealing, taboo search and others. They have in common that instead of constructing a best solution from scratch they start with an initial solution (found by some simple heuristic) and move around the adjacent search space looking for better solutions. The simplest example is the classic hill-climbing procedure, where we only continue on to near-by solutions that yield a better makespan. Disadvantages of this approach is that we do not have any means of emerging from a local optimum, as well as missing out on valuable information about the solution space explored so far that could be used for further improvements later on. Iterated hill-climbing beginning from different initial solutions often yields better results.

The advantage of local search procedures in general is that they incorporate some form of problem-specific knowledge that helps guide the procedure in the right direction. Problem-specific knowledge is very likely to be in the form of the definition of a *neighbourhood*; a solution x gives rise to a neighbourhood $N(x)$ comprising of solutions "close" to x. The definition of "close" is made using knowledge of the problem. Whenever moving from one solution x to another y, the move is made within the neighbourhood $N(x)$, meaning $y \in N(x)$. Various ways exist to how to choose the next neighbouring solution y: choosing the one that yields the best makespan, the one that yields the least deteriorating makespan or chosen at random, just to name a few.

In simulated annealing, for instance, a neighbour is chosen non-deterministically on the basis of probability. A neighbour is often chosen randomly and if the neighbour does not yield an improved makespan, it is accepted with a certain probability. Otherwise another neighbour is selected and tested again. This implies that in each step of the procedure we do not necessarily improve our makespan. We are, however, given an opportunity to escape from local optima, and the acceptance rate should be set so that it gets harder and harder to make this escape.

Taboo search is another of these local search strategies that incorporates yet another means of problem-specific knowledge: long- or short-term memory. Taboo search, like

simulated annealing, offers ways to escape from local optima, but deterministically: normally there is no element of probability involved in choosing the least deteriorating move to be performed next. The procedure prevents falling back into a local optimum by introducing a *taboo list*. This list contains the last $maxt$ moves performed, actively forbidding the repetition of these moves for the next $maxt$ iterations. This enables taboo search to wander away from a local optimum.

The type of list mentioned here is short-term memory, but as we shall see in section 4 long-term memory can also aid in guiding and restarting taboo search from previous good solutions.

2.5 Performance of Local Search Heuristics

The question arises: why are we at all interested in looking at methods that do not guarantee an optimal solution to JSSP when we indeed have the possibility to use exact methods, like branch and bound? Obviously, using exact methods on NP-hard problems gives us the answer within the definition of NP-hard: they cannot be solved in sub-exponential time. Although certain small and specific instances can be solved polynomially (as we saw in section 1.2), in general there are no computationally tractable methods for solving random JSSP instances to optimality. Therefore it is interesting to have a look at heuristics and how they perform compared to each other. In [1] the authors present a short but thorough comparison of different heuristics in JSSP: hill-climbing (also known as iterative improvement) with multiple starting points, simulated annealing, threshold accepting (accepting new solutions if their improvement to the makespan lies within a certain threshold) and genetic algorithms. They also use two different formulations of neighbourhoods, $N1$ and $N2$, which we shall return to shortly. Finally they compare these results to an implementation of taboo search.

Each type of heuristic is given the same amount of time to solve various instances of JSSP, allowing for a more "fair" comparison. Parameters are therefore adjusted to harmonise running times. Their results show that the basic hill-climbing heuristic is outperformed by most other heuristics, and that simulated annealing in general performs better than genetic algorithms and threshold accepting. Subsequently the authors allow the algorithms to take the time needed to finish. This last experiment confirms the strength of simulated annealing. But what about taboo search? Interestingly, this heuristic generally outperforms all other tested heuristics in many instances on both running time and quality of solution obtained.

Although the article is of some age (1994) it is interesting to see how an early implementation of taboo search has such great potential. In the next sections we shall go deeper into the nuts and bolts of taboo search and have a look at why neighbourhood definitions play such an important role.

3 Taboo Search

In this section we shall have a look at the core elements of taboo search: stopping criteria, the taboo list, aspiration functions and – most importantly – neighbourhoods.

Descriptions of the general method as well as most definitions will be short and based on [5], [7] and [9]; a more thorough explanation as well as examples can be found in [5].

Finally we have a look at the computational complexity of using taboo search to solve JSSP instances.

3.1 Definitions and Overview

As before mentioned, taboo search begins with an initial solution, found by some simple heuristic or insertion method. The algorithm then searches the *neighbourhood* of this solution (consisting of "similar" or "nearby" feasible solutions) for a better one. If one is found, the search then accepts this as the new initial solution and continues from here. Going from one feasible solution to another is called a *move*. In JSSP a move v will consist of switching two operations o_i, o_j from the set O in the current production plan π, e.g. $v = (o_i, o_j),\ i, j \in O$. We denote the new production plan obtained by applying move v to π by $Q(\pi, v)$. So far this procedure is basically the same as applying simple hill-climbing. Taboo search also maintains a *taboo list* $T \subseteq O \times O$ of length $maxt$ consisting of the previous $maxt$ moves made so far. In fact, instead of the actual move being stored, we store the reverse move v^{-1}: $v = (o_i, o_j) \Rightarrow v^{-1} = (o_j, o_i)$. The algorithm is forbidden to make any of the moves currently in the taboo list.

The taboo list is a form of short-term memory. Later we shall see how intermediate memory can be used to intensify search in "good" regions previously visited. Long-term memory can also be used to diversify the search into "interesting" regions that do not necessarily promise better solutions. In JSSP one can, for instance, keep a count on the edges that tells the algorithm how many times the edge has been used in a solution found. Long-term memory can then diversify the search by choosing solutions that do not include one or more of those edges with the largest count. Further diversification of the search can also be made by introducing probabilistic parameters in various stages of the search: accepting taboo moves with a certain probability (the aspiration function), making certain moves seem more attractive (redefinition of neighbourhood), and deciding whether to make used moves taboo or not (redefinition of taboo list properties). For more in-depth studies we refer to section 9 in [7]. The overall process of taboo search can be illustrated as in Figure 2 below.

Now we can also see the close relation to branch and bound algorithms: the taboo list in JSSP usually contains certain operation pairs (moves) used in previous iterations, but the list may also contain moves or operations with certain properties. Depending on the manner in which a move is defined, these may also consist of linear inequalities or logical relations. This is very much like the way a branch and bound algorithm would separate the solution space into smaller ones.

In case no better solution is found in the current neighbourhood, we have reached a local optimum and we choose "the best of the worst". How this is done varies from implementation to implementation, but it allows taboo search to progressively wander away from local optima instead of just cycling around them. Also, taboo search introduces an *aspiration function* that allows for certain taboo moves to be performed if for instance they yield a better makespan. In other words, the aspiration function defines exceptions for bringing in moves that were actually forbidden, but still retaining the ability to avoid cycling.

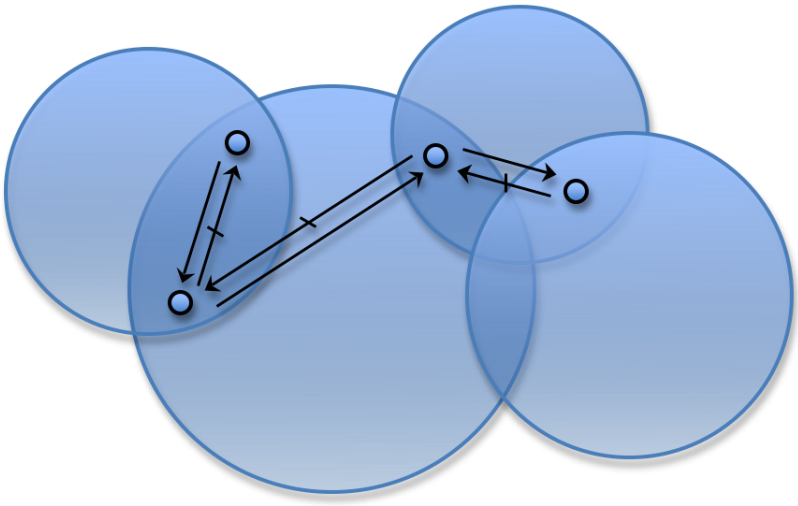

Figure 2: Rough sketch of 3 iterations of a TS algorithm. Normal arrows represent moves, slashed ones represent forbidden moves. Large blue circles indicate neighbourhoods belonging to the shown solutions.

3.2 Neighbourhood Definitions

Recall from section 1.1 that the makespan $C_{max}(\pi)$ associated with a given production plan π is equal to a (not necessarily unique) longest (critical) path in the corresponding graph $G(\pi)$. When discussing neighbourhood definitions it is necessary to further refine the division of a critical path, namely into blocks. A critical path u of length w can naturally be decomposed into the operations of which it consists:

$$u = (u_1, u_2, \ldots, u_w), \quad u_i \in O.$$

We shall now have a look at another important decomposition of a critical path, namely the division into *blocks*. A block is informally defined as the longest set of uninterrupted operations from a critical path on the same machine. The critical path then consists of one or more blocks, depending on the schedule at hand. Formally, for a critical path u containing r blocks we define a block B_j as a sequence of operations:

$$B_j = (u_{a_j}, u_{a_j+1}, \ldots, u_{b_j}), \quad j = 1, \ldots, r.$$

To ensure that the blocks actually make up the critical path when put together, we make sure that the indices are growing throughout the individual blocks as well as between the blocks:

$$1 = a_1 \leq b_1 < b_1 + 1 = a_2 \leq b_2 \cdots \leq a_r \leq b_r = w.$$

We require that operations in a block are processed on the same machine:

$$\mu(u_{a_j}) = \mu(u_{a_j+1}) = \cdots = \mu(u_{b_j}), \quad j = 1, \ldots, r$$

and that two consecutive blocks are processed on different machines:

$$\mu(a_j) \neq \mu(a_{j+1}), \quad j = 1, \ldots, r-1.$$

Example 2: *Returning to the instance from Example 1, we can construct a Gantt chart to visualise the processing order as well as the contained blocks:*

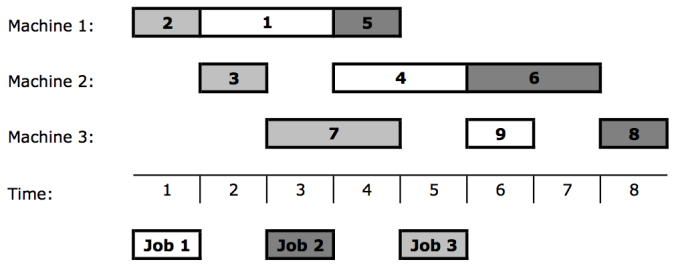

Figure 3: Gantt chart of the processing order π from Figure 1.

From the Gantt chart we can see that the critical path $u = (2, 1, 4, 6, 8)$ *consists of three blocks:* $B_1 = (2, 1)$, $B_2 = (4, 6)$ *and* $B_3 = (8)$. *We note that this processing order is not optimal. The talented reader should be able to find a makespan of 7.*

$\odot$

We shall now have a look at various ways of defining the neighbourhoods that are of such great importance in local search algorithms; some of these incorporate blocks. Neighbourhoods are often referred to as *operators*, as they are commonly defined as "what to do in order to reach a new solution". A few key concepts of neighbourhoods in general can be outlined:

Size The size of a neighbourhood is obviously of great importance: the larger the neighbourhood, the longer it takes for an algorithm to search it for the "best" solution.

Convergence We define convergence of a neighbourhood in terms of how well certain definitions of neighbourhoods help the algorithm converge towards local or global optima. In this paper we do not discuss this property.

Connectivity The connectivity property states that it is possible to get from any feasible solution to an optimal one using only those operations defined in the neighbourhood definition. In other words: from a given starting point one can always find a series of neighbourhood operations leading to an optimal solution. In the case of simulated annealing procedures, the connectivity property is required in order to ensure asymptotic convergence. Most of the neighbourhoods under investigation have this property. We shall see, though, that the most powerful one does actually not.

Neighbourhoods (or operators) are commonly named Nx with x being a positive integer. For examples on the uses of different neighbourhoods, we refer to [5]. Definitions,

sizes and remarks are based on [3] and [9]. We shall put our main focus on neighbourhoods *N1* and *N5*. *N5* is a subset of *N1* and therefore inherits some interesting properties. Also, little attention has been given to *N2*, *N3* and *N4* in the literature.

We begin with the simplest of the neighbourhoods: the *N1* neighbourhood, originally from [8].

N1: On a critical path in the current solution, reverse a disjunctive arc, i.e. an arc between two adjacent operations on the same machine.

In the notation of [9], if we denote by $V_1(\pi)$ all the solutions obtained by applying the *N1* operator to a solution π, let c be the number of critical paths, let r_i be the number of blocks on critical path i, and let $B_{k,i}$ be block k on critical path i, the *N1* neighbourhood can formally be written as

$$N1(\pi) = \{Q(\pi, v) : \ v \in V_1(\pi)\}$$

and the size of $N1(\pi)$ is then:

$$|N1(\pi)| = O\left(\sum_{i=1}^{c}\sum_{k=1}^{r_i}|B_{k,i}|\right).$$

If we assume that π has only a single critical path of length r, the size of the neighbourhood is naturally reduced:

$$|N1(\pi)| = O\left(\sum_{k=1}^{r}|B_k - 1|\right)$$

In [8] the authors prove that this neighbourhood is connected and contains only feasible solutions (assuming of course that one starts from a such).

N2: On a critical path in the current solution, consider the disjunctive arc (i, j) on machine m. Define $p(i)$ as the predecessor of i on machine m and $s(j)$ as the successor of j on machine m. Furthermore, restrict attention to arcs (i, j) at the beginning or end of a block, i.e. at least one of the arcs $(p(i), i)$ or $(j, s(j))$ is not on the critical path. Reverse (i, j). Let h be the direct predecessor of i in i's job and let k be the direct successor of j in j's job. Reverse the conjunctive arcs $(p(h), h)$ and $(k, s(k))$, if they yield an improvement in the makespan.

N3: On a critical path in the current solution, consider the disjunctive arc (i, j). With the definitions from *N2*, consider all permutations of the following operations, in which (i, j) is reversed: $\{p(i), i, j\}$ and $\{i, j, s(j)\}$.

N4: On a critical path in the current solution, consider an operation i in some block. Move i to the beginning or end of its block.

The neighbourhoods *N2*, *N3* and *N4* all have the connectivity property. They are, however, not quite as interesting as the *N5* neighbourhood:

N5: On a specific critical path in the current solution, consider swapping only those two successive operations that lie at the beginning or end of a block. For the first block consider only the two last operations, and for the last block consider only the two first operations.

The size of *N5* is trivially $|N5(\pi)| = O(r)$, i.e. the number of blocks on the critical path chosen. More precisely:

$$|N5(\pi)| = \min\{1, |B_1| - 1\} + \sum_{k=2}^{r-1} \min\{2, |B_k| - 1\} + \min\{1, |B_r| - 1\}$$

Contrary to other neighbourhood definitions, this one chooses arbitrarily only one critical path and does not take care of any others. This makes it computationally easier to find the critical path at hand instead of having to find them all. Also, the authors in [9] argue that this neighbourhood consists only of feasible solutions, as it is merely a subset of the *N1* neighbourhood:

$$N1(\pi) \supset N5(\pi).$$

They even prove that any processing order from *N1* that is not found in *N5* cannot yield a better makespan:

Theorem 1: *For any processing order* $w \in N1(\pi) \setminus N5(\pi)$ *we have that* $C_{max}(w) \geq C_{max}(\pi)$.

The proof is rather long and tedious and can be seen in [9], Appendix A. Note, however, that this neighbourhood does *not* have the connectivity property and is therefore not a good choice for simulated annealing procedures. Finally, the authors give a clear stopping criteria for the taboo search algorithm:

Theorem 2: *If* $V_5(\pi) = \emptyset$, *then* π *is optimal.*

Proof. Recall from the beginning of this section the property of a block on a critical path $u = (u_1, \ldots, u_w)$ of length w consisting of r blocks:

$$1 = a_1 \leq b_1 < b_1 + 1 = a_2 \leq b_2 \cdots \leq a_r \leq b_r = w.$$

From the definition of $N5(\pi)$ we have two possible cases in which the neighbourhood can be empty: all blocks have length 1, or there is only one block ($r = 1$). We have a look at the two cases separately:

1. We have that $a_j = b_j,\ j = 1, \ldots, r$. Therefore all edges on the critical path are from the set R, meaning that all operations in each block are from the same job $k \in J$. Therefore the size of the critical path is equal to the size of job k ($w = o_k$), and the critical path itself consist of the operations in job k ($u = (l_{k-1} + 1, \ldots, l_{k-1} + o_k)$). Also, the length of this critical path is equal to the total processing time of job k: $C_{max}(\pi) = \sum_{i=l_{k-1}+1}^{l_{k-1}+o_k} \tau_i$.
2. In the case with only one block B_1, we have that all operations on the critical

path are processed on the same machine. Denote this machine $q = \mu(a_1)$. The set M_q of operations processed on machine q therefore equals the operations on the critical path: $M_q = \{u_1, \ldots, u_w\}$, and the makespan equals the machine bound: $C_{max}(\pi) = \sum_{i \in M_q} \tau_i$.

□

The *N5* neighbourhood is one of the core foundations of the algorithm of Nowicki and Smutnicki. We shall later see that this neighbourhood has brought great results in their taboo search procedure.

3.3 Complexity of Taboo Search in JSSP

It seems interesting in this context to investigate exactly when taboo search is effective and when not. As we have seen in the previous section, a number of different neighbourhoods have been proposed for use in local search algorithms with varying results, and research is often focused on inventing new algorithms that decrease running times on certain comparable instances of JSSP. These instances are often referred to as being "exceptionally hard", but what actually makes an instance "hard" is a bit more blurry. Not much focus has been put on investigating these questions, however [13] brings some new approaches to the definition of "hard" instances.

Similar experiments have been made for the SAT (satisfiability) problem, but in [13] the authors – very conveniently – focus on taboo search in JSSP. They investigate what aspects of the solution space influence the actual running time of an implementation of taboo search and to what extent. The implementation used is by Taillard and can be found in [12]. The reasons for using this implementation are its simplicity due to using the *N1* neighbourhood and in addition it being fairly effective on a variety of instances. Also, using Taillard's implementation with the connectivity property of the *N1* neighbourhood guarantees us that the algorithm is, at least in theory, capable of locating optimal solutions to all instances when given enough execution time.

In order to investigate different search space properties, we need a function D to compute the "distance" between two solutions π_1 and π_2. The authors present the most widely used, neighbourhood-independent one: let π be some solution and let $i, j \in J$ and $k \in M$. We define the Boolean-valued operator $preceeds_{ijk}(\pi)$ to be true, whenever job i precedes job j on machine k in solution π. We then define the distance between solutions π_1 and π_2 as ([13], p9):

$$D(\pi_1, \pi_2) = \sum_{i=1}^{m} \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} preceeds_{ijk}(\pi_1) \text{ XOR } preceeds_{ijk}(\pi_2).$$

This function is normalised to get more smooth results between 0 and 1, independent of instance sizes:

$$0 \leq \overline{D}(\pi_1, \pi_2) = \frac{2D(\pi_1, \pi_2)}{mn(n-1)} \leq 1.$$

Different search space properties are then considered:

- $|optsols|$: the number of optimal solutions to the instance at hand.

- $|backbone|$: the size of the *backbone* in the instance at hand. The backbone is informally defined as features shared by *all* optimal solutions. In the disjunctive graph representation, this would correspond to comparing all optimal solutions and identifying those edges that are represented in all of these solutions. $|backbone|$ is then the number of these.

- $\overline{loptdist}$: the average distance between locally optimal solutions in the instance at hand. For two locally optimal solutions π_1 and π_2 this is formally defined as $\overline{D}(\pi_1, \pi_2)$.

- $d_{lopt-opt}$: the mean distance between local and global optimal solutions, again using $\overline{D}$ as the distance function.

Results are empirical: all experiments are conducted a large number of times on randomly generated small (6x4 and 6x6) instances of JSSP. We shall not delve deep into the statistical analyses of the article but merely discuss the results.

The authors test *static cost models* based on each of the four properties listed above. In other words, they generate 1000 instances of each of the groups and test 5000 runs on these instances to see if the property has a direct influence on the median processing time ($cost_{med}$) of the instances. The reason for doing this is the fact that Taillard's implementation has a certain amount of chance in it when deciding ties. Also, they investigate the magnitude of this influence. This way of investigating cost naturally requires a great knowledge of the instances at hand to find the given properties. Therefore the authors only consider small instances, as the number of local and global optimal solutions explodes when considering larger instances.

- $|optsols|$: when developing a static cost model based on the number of optimal solutions, the authors find this model to be fairly inaccurate. However accuracy is expected to increase as $n/m \to \infty$ (more rectangular instances). By intuition, though, one should think that a local search algorithm would spend more execution time when the number of optimal solutions is small. Given these results, one can not accurately say that the number of optimal solutions has any significant influence on the running time of a taboo search implementation.

- $|backbone|$: interestingly, the results of the static cost model based on backbone size are almost completely similar to those based on the number of optimal solutions. These two features can therefore be seen as equally unimportant in predicting running time.

- $\overline{loptdist}$: this model again shows itself to be fairly inaccurate. The authors do not spend much time on it, merely stating that the two previous models are at least as accurate as $\overline{lopdist}$.

- $d_{lopt-opt}$: the static cost model based on $d_{lopt-opt}$, on the other hand, shows a strong influence on the median running cost. The authors observe a much higher accuracy than those of the previous models considered, for both 6x4 and 6x6 instances. The model, however, becomes much more inaccurate when $cost_{med}$ (and consequently $d_{lopt-opt}$) increases. The authors conclude:

> *"[...] the static cost model based on $d_{lopt-opt}$ accounts for a substantial proportion of the variance in the cost required by $TS_{Taillard}$ to locate optimal solutions to 'typical' general JSPs."* ([13], p19)

As a final experiment the authors test the $d_{lopt-opt}$ cost model with the addition of one or more of the other three parameters. This, however, does not improve on the model.

Although these experiments are empirical and the implementation used is fairly old and simple, their results give a good indication of what factors play an important role in determining the "hard" instances of JSSP. Clearly, the mean distance between locally optimal and optimal solutions has *some* influence, whereas other tested parameters have shown to be uninteresting. In section 5 we shortly discuss how this knowledge can be utilised in future research.

4 Analysis of Two TS Algorithms

We are now ready to have a look at two state-of-the-art taboo search algorithms invented by Nowicki and Smutnicki. They are named TSAB and i-TSAB ("Taboo Search Algorithm with Back-Track Jumping" and "*Iterated* Taboo Search...") and first appeared in 1996 (see [9]) and 2001 (see [10]) respectively. i-TSAB builds on the foundations of TSAB and in this section we shall have a look at both algorithms – but with main focus on i-TSAB – and see why they are so effective. More detailed reviews of TSAB, including implementation notes and pseudocode, can be found in [5].

4.1 Short Review of TSAB and i-TSAB

TSAB is a fairly simple implementation of a taboo search algorithm. It uses the disjunctive graph representation and maintains a taboo list T of size $maxt$. Also, the *N5* neighbourhood is used for the first time in a local search algorithm. The authors initially present the algorithm TSA, which is a simpler version of TSAB. In TSA, firstly an initial solution is generated using the insertion technique from [15]: latin squares are used to iteratively generate a feasible schedule by inserting operations one at a time. We shall not put further attention to this method, as the authors in [9] experimentally verify that the quality of the initial solution does not have a substantial influence on the running time of their algorithm. Any heuristic for finding feasible solutions can be used at this stage.

From the initial solution found, TSA searches the *N5* neighbourhood for the best improving solution, uses this as the next starting point and appends the inverse move to the taboo list. If there are no unforbidden improving moves, the algorithm searches the taboo list for a forbidden move that yields a lower makespan. If one is found, this move is applied. This is the aspiration criteria in TSA. If no improving solution can be found, TSA selects the "best of the worst" from the taboo list in the following way: re-append the last element of T to T just as many times as required for $V_5(\pi) \setminus T$ to be non-empty, e.g. the "oldest" move is "pushed out" of T, losing its taboo status.

TSA continues until it has either found an optimal solution (for instance by Theorem 2), has found a solution "close enough" to the optimal one, or has exceeded some limit on running time or iterations without improvement.

TSAB additionally introduces long-term memory: it maintains a list L of the previous *maxl* best solutions found. The entries of the list contain the production plan, the associated taboo list, and the move performed directly after. When the run of TSA from the initial solution has finished, it restarts from the first entry in L and "reloads" the situation, continuing with a different move than before, until L is empty. Also, TSAB has long cycle detection that aborts a certain search path, if the algorithm keeps circling around some part of the solution space. It merely keeps track of makespan values during the course of some number of iterations.

i-TSAB is a more advanced and refined version of TSAB that still, though, has TSAB as a core sub-routine. First *i*-TSAB generates a pool E of initial solutions. The first solution e_1 is generated by running TSAB on the heuristic initial solution e_0 generated by the insertion technique from [15]. The remaining solutions in the pool are generated by applying a so-called *path relinking procedure* between two previous solutions, as described in [10]. This approach resembles the line of thought used in genetic algorithms: combining parts of critical paths taken from different initial solutions.

The approach of the next part of *i*-TSAB relies on the firm belief that a "big valley" phenomenon exists in the JSSP solution space: the distances between good local optimal solutions and the globally optimal ones are positively correlated with the makespan values of these solutions. In other words, the best solutions available are "close" to each other. On the basis of this belief, *i*-TSAB enters its "proper work" phase. In this phase the pool of solutions generated above is explored using TSAB in the following way: the solution from the pool with the best (lowest) makespan e^* is found. Additionally, the solution e^x that is furthest away – in terms of the number of moves required to transform it into e^* – is also found. Using another path relinking procedure, a solution ϕ "halfway between" e^* and e^x is found. Then TSAB is run from ϕ to find a new solution e^c which unconditionally replaces e^x in E: $E = (E \setminus \{e^x\}) \cup \{e^c\}$. This procedure continues until the distance between e^* and e^x is below some specified bound. The procedure is outlined in Figure 4.

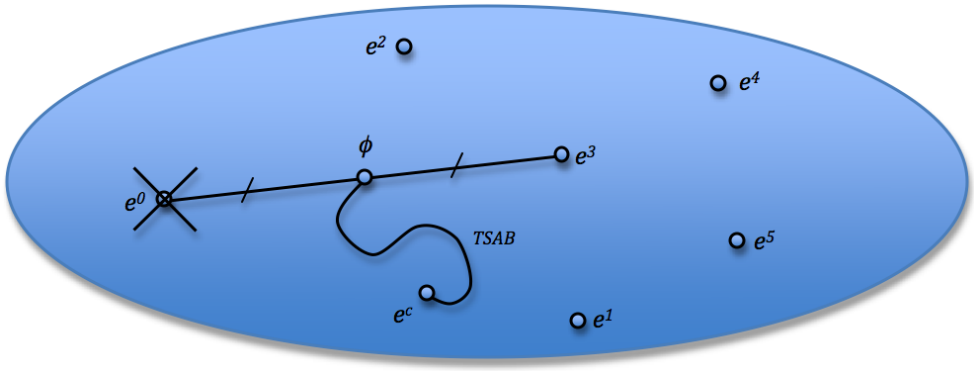

Figure 4: Rough sketch of the "proper work" phase of *i*-TSAB. Shown is the pool E of initial solutions. $e^* = e^3$ has the lowest makespan in the pool, and $e^x = e^0$ is furthest away from e^3, with solution ϕ "halfway between" them. TSAB is then applied to ϕ resulting in a new solution e^c that unconditionally replaces e^0.

So basically *i*-TSAB works by generating a pool of good solutions and using the big

valley conjecture to close in on the good solutions that are supposed to be close to the ones in the pool.

4.2 Empirical Analysis of i-TSAB

As can be seen in section 4 in [9], the TSAB algorithm, when it first appeared in 1993, significantly outperformed three of the other best-known approximation algorithms at the time by a large margin. Not only did it find lower makespans in many cases, it also drastically reduced the computational effort needed to achieve these results. The latter can specifically be seen in Table 2 in [9].

i-TSAB further improved on makespans and execution time in 2003 ([10]). They summarise their results by stating that out of 112 previously unsolved instances they manage to find 91 better upper bounds.

But what actually makes i-TSAB work so well compared to other approximation algorithms? In [14] the authors conduct a thorough study of the i-TSAB algorithm to reveal what components of the algorithm make it perform so well. They set up a framework to test basic metaheuristics in comparison with taboo search: a refined version of iterated local search based on the *N5* operator incorporating cycle detection, and a Monte Carlo-based algorithm much like simulated annealing but with fixed temperature, again based on the *N5* operator. The taboo search algorithm to be compared with is a stripped-down version of i-TSAB, lacking long-term memory and the more complex pool of initial solutions. In general, the stripped-down version resembles the TSA procedure in most ways.

Results are surprising: in a general environment where *N5* is used throughout the line-up of metaheuristics, no specific advantage can be seen for taboo search in general. The two other metaheuristics obtain equally good results when using the *N5* operator. Still, though, i-TSAB outperforms them all by a large margin. The conclusion then is that taboo search by itself cannot explain the advantages of the i-TSAB algorithm. However, the authors are surprised to see such strong results from relatively simple metaheuristics employing the *N5* operator.

Next the authors "augment" the metaheuristics by adding long-term memory, one of the classical core components of general taboo search. This step significantly improves almost all of the algorithms considered, yet they are still not quite as good as i-TSAB. This also adds more belief to the hypothesis that taboo search in itself is not necessarily the best metaheuristic for the JSSP.

Finally the authors adjust the way and frequency with which the algorithms perform intensification and diversification. Remember that in i-TSAB intensification is performed by applying TSAB to the solution ϕ located between e^* and e^x from the pool E. Diversification is in the form of looking at two new solutions in the pool. The authors have a slightly different take on intensification and diversification in their augmented algorithms (see [14], section 5), as this allows them to control the amount of each and monitor the impact on different metaheuristics.

In the above experiment on adding long-term memory, the authors allowed the algorithms an equal amount of intensification and diversification. In this section they adjust these amounts to investigate whether this has an effect on the efficiency of the algorithms.

They find that focusing solely on one of the methods generally yields worse solutions than applying a mixture of the two. Also, based on their empirical observations they hypothesise a "sweet spot" when using a straightforward 50/50 split between intensification and diversification.

Summing up the above observations, the authors conclude that

> *"the core metaheuristic, and in particular tabu search, is* not *integral to the performance of i-TSAB. Rather, i-TSAB achieves its state-of-the-art performance levels through the use of both the N5 move operator and a balanced combination of intensification and diversification."* ([14], p25)

5 Further Improvements and Unexplored Terrain

Having now gone through the nuts and bolts of local search heuristics, taboo search, state-of-the-art algorithms, complexity and empirical results, we now turn our focus to a short discussion of where research might go in the near future.

Actually, many attempts have already been made to develop enhanced methods for the replacement of i-TSAB. For instance, in [16] the authors present a new neighbourhood structure *N6* which they apply to fairly simple taboo search. Also, they use simulated annealing instead of the insertion heuristic of [15] to generate a starting feasible solution. They actually manage to improve on the results of i-TSAB on various instances, however they conjecture that i-TSAB is still unbeaten on instances where $n \leq 2m$.

In the literature there is still a tendency towards trying to improve neighbourhood definitions or combining different search strategies in new and creative ways to obtain better results. Since i-TSAB, however, none of these attempts have shown substantial improvements. As we saw in sections 3.3 and 4.2, there are still sparse results on what actually makes a good taboo search algorithm, just as the JSSP solution space could very well reveal more interesting properties.

Another way to improve on the basics of the algorithms without inventing new and even more complex neighbourhoods could be using more efficient data structures and ways to do the massive amounts of calculations performed by almost all algorithms today. Looking at data structures, representing the disjunctive graph is usually done in various classic ways: neighbourhood matrices, predecessor lists, and successor lists. It is mostly unclear from the literature what representation is used, as apparently little focus is put on this subject. In [2] the authors propose a new and improved, compact representation of the disjunctive graph model: the graph matrix. This matrix of size $O(n^2)$ has all the benefits and properties of the three classic representations mentioned before, while improving on the complexity of some of the operations performed on the matrix. Experiments conducted using the graph matrix show remarkable decreases in running time compared to the classic representations.

6 Conclusions and Remarks

In this paper we have presented the job-shop scheduling problem (JSSP) and briefly outlined its complexity, notation and representation. We have looked at various solution techniques and compared these with each other as well as the highly efficient taboo search metaheuristic. We have seen preliminarily that taboo search outperforms other traditional heuristics as well as coming closer and closer to optimal solutions, as new algorithms are developed.

Attention has been brought to the fact that determining what factors influence the "difficulty" of certain JSSP instances is notoriously difficult to say anything about, except for the number of optimal solutions.

Also, we have had a thorough look at neighbourhood definitions and their importance, especially in taboo search. We saw that the *N5* neighbourhood is a strong reason for the success of more recent TS implementations, two of which we took an in-depth look at: the TSAB and i-TSAB algorithms of Nowicki and Smutnicki.

Based on thorough computational experiments, it has been outlined that i-TSAB gets its strength from both the *N5* neighbourhood as well as a balanced use of intensification and diversification. Finally, we have briefly outlined further possible research of taboo search in a JSSP context.

References

[1] E.H.L. Aarts et al: *"A Computational Study of Local Search Algorithms for Job Shop Scheduling"*, in *ORSA Journal on Computing*, Vol.6, No. 2, p118-125 (1994).

[2] J. Blazewicz, E. Pesch, M. Sterna: *"The disjunctive graph machine representation of the job shop scheduling problem"*, in *European Journal of Operational Research 127*, p317-331 (2000).

[3] J. Blazewicz, W. Domschke, E. Pesch: *"The job shop scheduling problem: Conventional and new solution techniques"*, in *European Journal of Operational Research 93*, p1-33 (1996).

[4] P. Brucker: *"An Efficient Algorithm for the Job-Shop Problem with Two Jobs"*, in *Computing*, Vol.40, No. 4, p353-359 (1988).

[5] U.G. Christensen, A.B. Pedersen, K. Vejlin: *"A Fast Taboo Search Algorithm for the Job Shop Scheduling Problem"*, lecture note in the course "Optimisation Problems in Production Planning", Department of Computer Science, University of Copenhagen (2008). Available from: http://www.andersbp.dk/dat/OPPP/taboo.pdf.

[6] M.R. Garey, D.S. Johnson, R. Sethi: *"The Complexity of Flowshop and Jobshop Scheduling"*, in *Mathematics of Operations Research*, Vol.1, No.2, p117-129 (1976).

[7] F. Glover: *"Taboo Search - Part I"*, in *ORSA Journal of Computing*, Vol.1, No. 3, p190-206 (1989).

[8] P.J.M van Laarhoven et al: *"Job Shop Scheduling by Simulated Annealing"*, in *Operations Research*, Vol.40, No. 1, p113-125 (1992).

[9] E. Nowicki, C. Smutnicki: *"A Fast Taboo Search Algorithm for the Job Shop Problem"*, in *Management Science*, Vol. 42, No. 6, p797-813 (1996).

[10] E. Nowicki, C. Smutnicki: *"An Advanced Taboo Search Algorithm for the Job Shop Problem"*, in *Journal of Scheduling*, Volume 8, Issue 2, p145-159 (2005).

[11] B. Roy, B. Sussmann: *"Les problèmes d'ordonnancement avec constraintes disjonctives"*, SEMA, Note D.S., No. 9 (1964).

[12] E.D. Taillard: *"Parallel Taboo Search Techniques for the Job Shop Scheduling Problem"*, in *ORSA Journal on Computing*, Vol.6, No. 2, p108-117 (1994).

[13] J.P. Watson, J.C. Beck, A.E. Howe, L.D. Whitley: *"Problem Difficulty for Tabu Search in Job-Shop Scheduling"*, in *Artificial Intelligence*, 143(2), p189-217 (2003).

[14] J.P. Watson, A.E. Howe, L.D. Whitley: *"Deconstructing Nowicki and Smutnicki's i-TSAB Tabu Search Algorithm for the Job-Shop Scheduling Problem"*, in *Computers & Operations Research*, Volume 33, Issue 9, p2623-2644 (2006).

[15] F. Werner, A. Winkler: *"Insertion Techniques for the Heuristic Solution of the Job Shop Problem"*, in *Discrete Applied Mathematics*, Volume 58, p191-211 (1995).

[16] C.Y. Zhang, P. Li, Y. Rao, Z. Guan: *"A very fast TS/SA algorithm for the job shop scheduling problem"*, in *Computers & Operations Research 35*, p282-294 (2008).