A Fast Taboo Search Algorithm for the Job Shop Scheduling Problem

Uffe Gram Christensen (uffe@diku.dk)
Anders Bjerg Pedersen (andersbp@diku.dk)
Kim Vejlin (vejlin@diku.dk)

October 21, 2008

Abstract: In this lecture note we initially present the Job Shop Scheduling Problem (JSSP) in a slightly different context and re-introduce the notions of critical paths and blocks. We then move on to describing an algorithm for obtaining near-optimal solutions to JSSP's, based on a local search strategy called Taboo Search. Taboo Search makes extensive use of neighbourhoods and it is of great importance to the algorithm to limit the size of these neighbourhoods, as this lecture note will focus on. We will ease reading by extensive use of examples.

1. A Fast Taboo Search Algorithm

1.1. Problem definition

The problem we are looking at is the Job Shop Scheduling Problem or JSSP. In this problem we have a set of jobs, J. Each job consists of a number of operations. The set of all operations is called O. We also have a set of machines, M, on which the individual operations can be processed. The size of each of the three sets will vary with the specific instance of the problem we are working with, but for the sake of generality we will be referring to the size of J as n, the size of O as o, and the size of M as m. A summarised description of the introduced variables is given in Appendix A.1.

Our object is to minimise the makespan of the jobs, that is to minimise the time it takes to complete all the jobs.

In order to illustrate our points better, we will now introduce an example to support this notation. In our example we have 3 jobs ($n = 3$), 2 machines ($m = 2$), and 12 operations ($o = 12$). See also Figure 1.

As indicated by the example, the jobs don't necessarily have the same amount of operations, but we will number the operations in O so that operations of the same job are next to each other in the set. We will also order the operations of a single job, so that the operations that are to be performed earlier are indexed with a lower number than operations that are to be performed later. If, for instance, the first job has 7 operations and the second job has 3 operations, then the operations of job 1 would be numbered 1

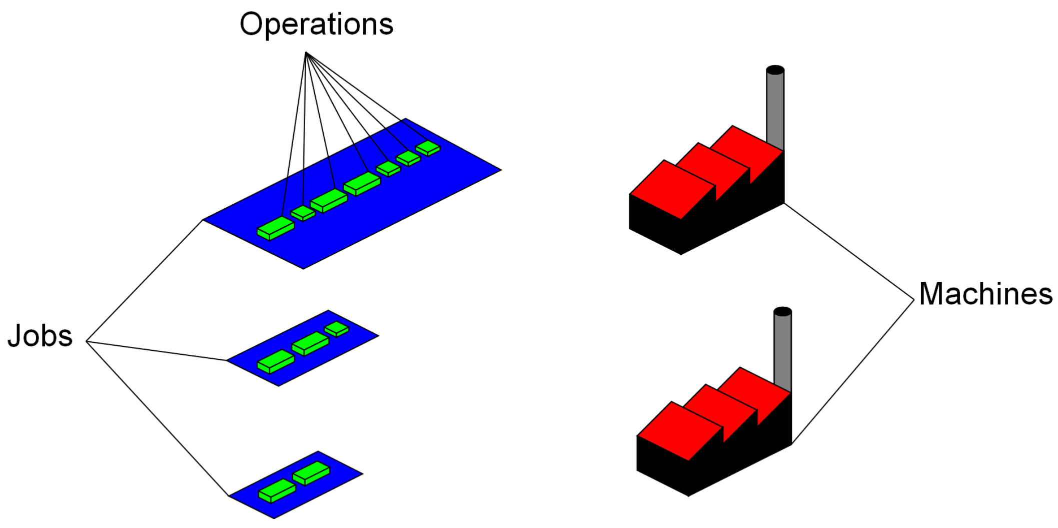

Figure 1: A JSSP example

through 7 and the operations of job 2 would be numbered 8 and 9. Operation 1 would then have to be performed prior to any of the operations 2 through 7, but not necessarily prior to operations 8 or 9. We will be using the variable o_i to indicate the number of operations in job i and define the variable $l_j = \sum_{i=1}^{j} o_i$ as the number of operations in jobs 1 through j. In the following table we present the value of these variables in our example:

Job (i)	1	2	3
o_i	7	3	2
l_i	7	10	12

Each operation has a specific machine it must be completed on and an amount of time it takes to complete the operation. The operation cannot be interrupted during this time period (i.e. no pre-empting). For the i'th operation we denote the relevant machine by $\mu_i \in M$ and the time it takes to complete by $\tau_i > 0$. Below we have a table with the relevant numbers of τ_i and μ_i for our example:

Operation (i)	1	2	3	4	5	6	7	8	9	10	11	12
μ_i	1	2	1	2	1	2	1	1	2	1	1	2
τ_i	2	1	2	2	1	1	1	2	2	1	2	2

As can be seen from the table, successive operations of a job are processed on different machines. We will make this a requirement for all JSSP instance, but it isn't hard to transform a JSSP instances that doesn't have this property into one that does; we simply interject an artificial operation requiring an infinitesimal amount of time. This artificial operation is performed on a different machine.

We now define the variable S_i as the time we initiate processing of an operation, this means that the makespan can now be formulated as $\max_{i \in O}(S_i + \tau_i)$. The makespan is the span of time between the initiation of the first operation and the conclusion of the final operation.

1.2. Representing the problem as a graph

The following definitions will come in handy when we try to represent the problem as a graph, thereby facilitating our analysis.

We define the set $M_k = \{i \in O : \mu_i = k\}$ that is the set of all operations using machine k. We define the size of this set as $m_k = |M_k|$. We can now talk about the processing order of operations on a machine using these variables. Ordering the elements of M_k by some random method, we obtain a permutation which we shall call π_k. If we index the elements of π_k, the permutation would look like this $(\pi_k(1), \ldots, \pi_k(m_k))$. $\pi_k(i)$ is thus the i'th element of π_k and naturally an element of M_k. Many different permutations exist and so we let Π_k denote the set of all different permutations, π_k. The actual processing order of all operations is thus given by a permutation $\pi = (\pi_1, \ldots, \pi_m)$ where $\pi \in \Pi = \Pi_1 \times \Pi_2 \times \cdots \Pi_m$.

For a specific processing order π, we now construct the directed graph $G(\pi) = (O, R \cup E(\pi))$ where each operation constitutes a node and the set of edges is defined by

$$\begin{aligned} R &= \bigcup_{j=1}^{n} \bigcup_{i=1}^{o_j - 1} \{(l_{j-1} + i, l_{j-1} + i + 1)\} \\ E(\pi) &= \bigcup_{k=1}^{m} \bigcup_{i=1}^{m_k - 1} \{(\pi_k(i), \pi_k(i+1))\} \end{aligned}$$

The arcs represented by the set R are arcs between operations in a job. Each operation has an arc to the next operation in the job, thus the last operation of the job has no such arc. For those who have read [1] these are also known as the *conjunctive edges*. The arcs represented by the set $E(\pi)$ are the arcs between operations on the same machine, that is, operations belonging to the same M_k. Each operation in M_k has an arc to the operation succeeding it as defined by the permutation π_k. In [1] these are known as the *disjunctive edges.*

Returning to our example we select a permutation of operations π which in this case is also a feasible order of production.

$$\begin{aligned} \pi &= (\pi_1, \pi_2) \\ \pi_1 &= (1, 8, 3, 11, 5, 10, 7) \\ \pi_2 &= (2, 4, 9, 12, 6) \end{aligned}$$

This permutation combined with the information about the jobs we have from earlier, gives us Figure 2, where the solid edges are the conjunctive edges and the dashed lines are the disjunctive edges.

In this directed graph we now let each node have a weight equal to the processing time of the node, that is τ_i, while the edges have a weight of 0. By finding a longest path to a node, we determine the earliest starting time of the corresponding operation, when using the selected permutation. This only holds if there are no cycles in the graph. [3] states that if a feasible production schedule is used, then the graph will contain no cycles. Since the longest path to a node determines the earliest starting time of the corresponding operation, the problem of finding the makespan is now reduced to finding the longest, or *critical*, path in the graph. Since the graph has no negative weight cycles,

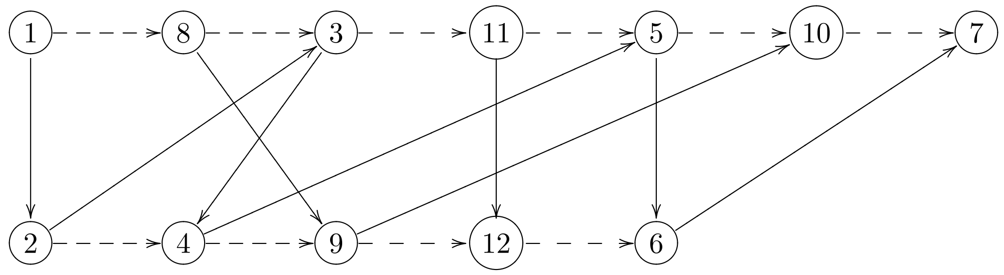

Figure 2: Graph of the 3-job, 2-machine, 12-operations instance.

this can be done by multiplying all node weights by -1 and finding the shortest path using Dijkstra's algorithm.

Such a critical path, u, can be written as the sequence of nodes it visits: $(u_1, \dots, u_w)$, $u_i \in O, i \in [1, w]$. This path can be divided into blocks, where each block is a number of operations performed in sequence on the same machine. Also, each block has the maximum length possible, meaning that a new block doesn't start unless we begin processing on a different machine. More formally we denote a block B_j and define it as $B_j = (u_{a_j}, u_{a_j+1}, \dots, u_{b_j})$ where the indices are growing in the block and the indices of block $j+1$ are greater than those of block j or put more mathematically $1 = a_1 \leq b_1 < b_1 + 1 = a_2 \leq b_2 < b_2 + 1 = a_3 \leq \cdots \leq a_r \leq b_r$. The reason for the weak inequalities as opposed to strict inequalities are from the fact that a block may have length one resulting in $a_j = b_j$. As we mentioned we require all operations in a block to use the same machine, that is $\mu(B_j) \overset{def}{=} \mu_{u_{a_j}}$, $j = 1, \dots, r$. Also, we require that the blocks contain all operations in a sequence that are processed on the same machine: $\mu(B_j) \neq \mu(B_{j+1}), j = 1, \dots, r-1$. In Figure 3 we show an example of how these blocks are defined in our example. As can be seen the blocks only extend along the critical path.

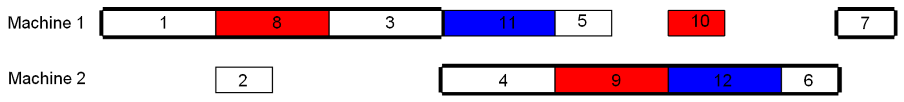

Figure 3: An example of blocks on the critical path.

Now you might ask what all this is good for, and that is indeed a very good question. The blocks will be used in the taboo search, which will be explained in detail later. They will also play a role in defining the neighbourhoods used in the taboo search.

1.3. Taboo Search in General

In this section we will outline a meta-heuristic approach to finding near-optimal solutions to certain optimisation problems. It will also illustrate the basic concept of this note: taboo search. Taboo search (TS) is a local search strategy that takes as input some solution to a given optimisation problem, found by some heuristic method, and goes on to look for better solutions in the *neighbourhood* of the given solution. It reaches this new solution by means of a *move* that takes the algorithm to a new and hopefully improved solution. This strategy continues until a satisfying near-optimal, or sometimes even optimal, solution has been found. Other *stopping rules* are also applicable. An illustration of the flow of the algorithm can be seen in Figure 4 below. Let us now delve deeper into the general algorithm.

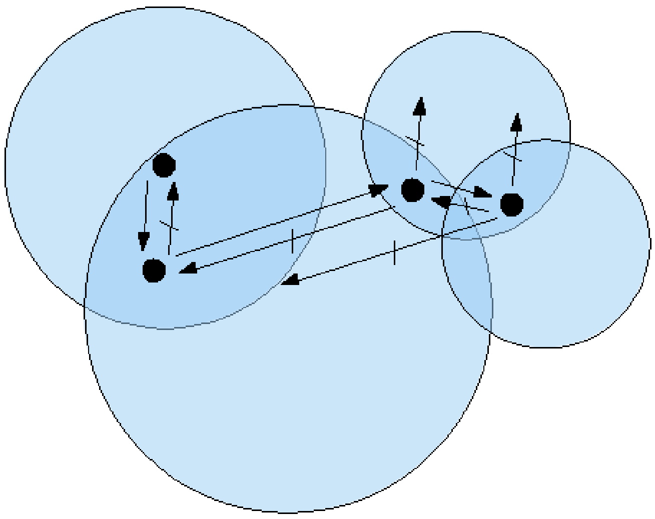

Figure 4: Rough sketch of 3 iterations the general TS algorithm. Normal arrows represent moves, slashed ones represent forbidden moves. Blue circles indicate neighbourhoods belonging to the shown solutions, marked with a black dot.

Our initial solution leads us to define a function, from now on called a *move*. A move transforms one feasible solution into another, for instance by interchanging two arcs used in a graph in TSP or switching the order of two operations in JSSP. Every move leads to a new solution that may be better or worse than the original. The subset of *applicable moves* (in this context defined as the set of moves leading to another feasible solution) generates a set of new solutions, which we will refer to as the *neighbourhood* of the original solution. TS now tries to find the best possible solution in this new neighbourhood by some searching strategy, on which we will take a closer look in section 1.10. The new-found solution will then serve as the initial solution in the next step of the algorithm.

The above does not seem like an efficient nor solid strategy for obtaining near-optimal solutions. However, TS uses certain memory techniques to improve search times and prevent cycling. Firstly, the algorithm only chooses a new solution from the neighbourhood that is at least as good as the previous one (if such a corresponding move exists). Otherwise it uses some strategy to select the "best of the worst" moves. More on this later. Whenever a move is selected and thereby leading us to a new solution, the *inverse move* (i.e. the move leading back to the solution that we came from) is added to a list containing "forbidden" moves. If the list is full, we delete the oldest entry. The length ($maxt$) of this *taboo list* often has substantial influence on the running time of the algorithm. The usage of this type of list is commonly referred to as *short-term memory*, however implementations using *long-term memory* may also be useful in a variety of situations.

We may encounter a situation, in which a forbidden move actually improves the solution value without making a cycle. To accommodate these instances we define an *aspiration function* that helps us evaluate the profit gained by making a forbidden move. Usually, such a move is allowed if it results in a solution value better than all the previously obtained solutions.

Several rules can determine whether to stop iterating the search or not:

- The search has found a solution that is within a certain given interval of the lower bound of the objective function value (i.e. we are "close enough" to the optimal solution).
- The search has not been able to improve the best solution during some number of fixed iterations (i.e. it has not improved during *maxiter* iterations).
- The search has exceeded the amount of time (CPU or real) allocated for solving the problem.

We shall take a closer look at these criteria in section 1.10.

1.4. Neighbourhoods

Taboo search builds on a number of central elements that need to be defined for the specific problem. The first of these is the *transition* or *move* operation, which modifies one feasible solution to produce a new solution. We looked at this definition in the above section.

The second definition needed is a more concrete definition of a neighbourhood, as also mentioned in the above section. In [3] various neighbourhoods are defined based on some subset of all possible moves. These neighbourhoods (and an additional neighbourhood) will be described in more detail in sections 1.5 through 1.8. The definition of a new neighbourhood is the major contribution of [3]. In the article the authors define a comparatively small neighbourhood and then show that it is probable that this neighbourhood definition will perform favourably compared to previous neighbourhood definitions.

1.5. Neighbourhood definition 1

Let $V'''(\pi)$ denote the set of all operation-pair interchanges where both operations are performed on the same machine. Then the neighbourhood $H'''(\pi)$ is defined as

$$H'''(\pi) = \left\{Q(\pi, v) : v \in V'''(\pi)\right\},$$

where $Q(\pi, v)$ is the operating order produced by applying move v to operating order π.

The size of the neighbourhood is quadratic:

$$|H'''(\pi)| = \Theta\left(\sum_{k=1}^{m} m_k^2\right)$$

where m is the number of machines and m_k is the number of operations on machine k. This neighbourhood may (and will for non-trivial problem sintances) also contain infeasible operating orders, and in each iteration all operating orders in $H'''(\pi)$ need to be generated and checked for feasibility.

Example 1. *Revisiting the example with* $\pi = (\pi_1, \pi_2)$, $\pi_1 = (1, 8, 3, 11, 5, 10, 7)$ *and* $\pi_2 = (2, 4, 9, 12, 6)$*, the graph* $G(\pi)$ *can be plotted:*

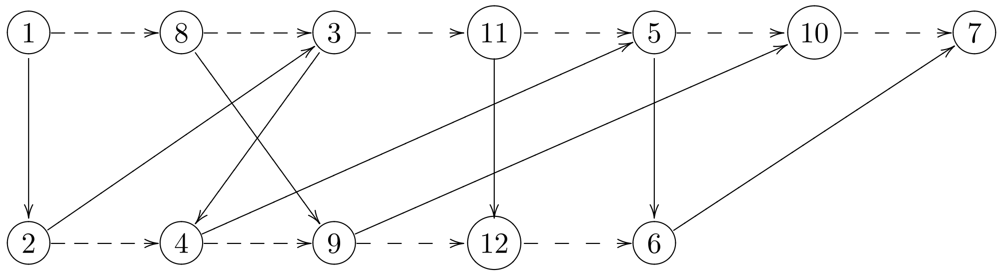

Figure 5: $G(\pi)$

The neighbourhood $H'''(\pi)$ *is then the set of processing orders arrived at by applying the following moves to* π*:*

$$(1,8),(1,3),(1,11),(1,5),(1,10),(1,7),(8,3),(8,11),\ldots,(9,12),(9,6),(12,6)$$

Some of these moves result in infeasible processing orders, because they violate the intra-job ordering of operations. One example of a move that generates an infeasible processing order is the move $(1,3)$*. Interchanging operations* 1 *and 3 results in operation* 3 *being processed before operation* 1*. This is not compatible with the requirement that operation* 1 *must be processed before operation* 2 *which in turn must be processed before operation* 3*. That* $Q(\pi,(1,3))$ *is infeasible can be seen by computing the graph* $G(Q(\pi,(1,3)))$ *and noticing that the cycle 3-8-1-2-3 has been introduced (see Figure 6).*

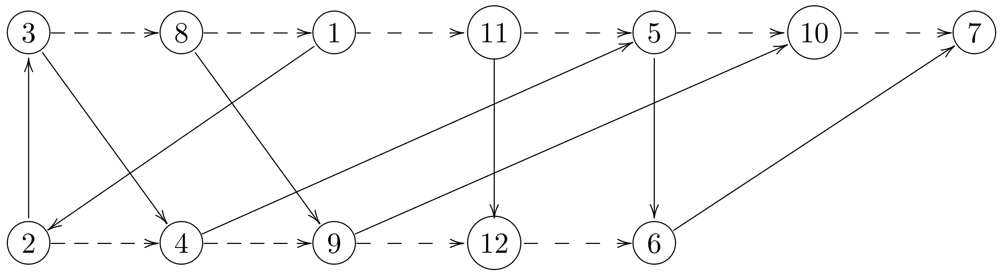

Figure 6: $G(Q(\pi,(1,3)))$ introducing the cycle 3-8-1-2-3.

It is clear that for any feasible operating order, any other feasible operating order can be reached through a series of job-pair interchanges. The implication is that an optimal processing order is always reachable from any starting point for a search. This is called the connectivity property.

1.6. Neighbourhood definition 2

Restricting the number of legal moves to only operation-pair interchanges involving adjacent operations, will significantly reduce the size of the neighbourhood while retaining the connectivity property.

Let $V''(\pi)$ denote the set of all operation-pair interchanges, where the operations are adjacent in the ordering on the same machine. The neighbourhood $H''(\pi)$ can then be defined as:

$$H''(\pi) = \left\{ Q(\pi, v) : v \in V''(\pi) \right\}.$$

Thus, restricting the number of legal moves results in a neighbourhood of smaller size:

$$|H''(\pi)| = \Theta\left(\sum_{k=1}^{m} (m_k) \right),$$

where m_k is the number of operations on machine k.

To see that the connectivity property still holds, observe that any operation pair interchange can be achieved through a series of pair interchanges involving only adjacent pairs. It also follows that $H''(\pi)$ may contain infeasible processing orders.

Example 2. *For our example* $\pi = (\pi_1, \pi_2)$*,* $\pi_1 = (1, 8, 3, 11, 5, 10, 7)$*, and* $\pi_2 = (2, 4, 9, 12, 6)$*. The set of moves is:*

$$(1,8), (8,3), (3,11), (11,5), (5,10), (10,7), (2,4), (4,9), (9,12), (12,6)$$

As mentioned some moves may violate the intra-job ordering restrictions. An example is the move $(2, 4)$*.*

1.7. Neighbourhood definition 3

The set of moves can be further restricted by in addition to the above requirements also requiring that the operation pairs be on some critical path.

This is the neighbourhood definition used in [2]. The size of this neighbourhood $H'(\pi)$ is

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

where c is the number of critical paths in the graph $G(\pi)$, r_i is the number of blocks on critical path i and $\mid B_{k,i} \mid$ is the number of operations in block k on critical path i.

It has been shown that this neighbourhood definition retains the connectivity property, and that $H'(\pi)$ contains only feasible processing orders. $H'(\pi)$ also contains only feasible processing orders. Neighbourhood $H'(\pi)$ is significantly smaller than $H''(\pi)$ and there is no need to test processing orders for feasibility. The neighbourhood definition does however require that all critical paths be found, which is equivalent to solving finding all shortest paths between a pair of nodes in a graph. This is generally a non-trivial problem.

Example 3. *For our example there exists only a single critical path* $1-8-3-4-9-12-6-7$*. This results in the set of moves* $(1,8), (8,3), (4,9), (9,12), (12,6)$*.*

1.8. Neighbourhood definition 4

In [3] the authors restrict the size of the neighbourhood $H(\pi)$ even further by only allowing moves on operation-pairs at the boundary of the blocks of a single critical path. More precisely only the first and last operation-pairs of a block are considered legal moves. The first and last blocks are treated differently in that for the first block only the last operation-pair is considered and for the last block of the selected critical path only the first operation-pair is considered. The reason for treating these two blocks differently is unclear to us.

This approach results in a neighbourhood size of $O(r)$ where r is the number of blocks on the critical path, and m is the number of machines.

The authors argue briefly that this neighbourhood is interesting because the set $H'(\pi)\backslash H(\pi)$ contains only processing orders that are not promising, based on a proof that

$$C_{max}(\alpha) \geq C_{max}(\pi) : \alpha \in H'(\pi)\backslash H(\pi)$$

where $C_{max}(\pi)$ is the makespan of processing order π. In addition to the neighbourhood being very small it is also worth noting that for $H(\pi)$ only a single critical path needs to be found (can be done with a modified Dijkstra algorithm), and as with neighbourhood $H'(\pi)$ all processing orders in $H(\pi)$ are feasible.

Those attributes are attractive since they lead to a fast search algorithm. There is however a price and it is briefly mentioned in the article that the connectivity property does not hold for neighbourhood $H(\pi)$. This shows that non-promising solutions are sometimes needed to get to more promising solutions, and that in some instances, using neighbourhood $H(\pi)$ is guaranteed to make it impossible to locate an optimal solution.

Example 4. *Returning to the example yet again, the set of legal moves (using neighbourhood definition $H(\pi)$) is:* $(1, 8), (8, 3), (4, 9), (12, 6)$. *While the reduction in number of moves is small for this example compared to neighbourhood definition 3, this is not generally the case. In cases with more than one critical path and where blocks are long the difference will be significant.*

1.9. Taboo Search List and Implementation Details in JSSP

We now return to the more practical issues of implementing a TS algorithm. For our implementation of the TS algorithm we construct and maintain a taboo list $T = (T_1, \ldots, T_{maxt})$ of length $maxt$ (decided by the current JSSP problem at hand), where each element of T consists of a valid move $T_j \in O \times O$, $j = 1, \ldots, maxt$. The list is initially set to contain only zero-moves, i.e. at first

$$T := (\underbrace{(0,0), (0,0), \ldots, (0,0)}_{maxt \text{ elements}}).$$

We add a move $v = (x, y) \in O \times O$ by shifting all elements of T to the left and appending v to the right end of the list. The element at position 1 is removed from T. We denote the insertion of v in T by $T \oplus v$. By implementing T as a circular list, we can perform this insertion in $O(1)$ time.

Whenever a valid move $v = (x, y)$ is performed in TS, we add the forbidden *inverse move* $\bar{v} = (y, x)$ to T, i.e. $T := T \oplus \bar{v}$.

Example 5. *Assume that at some iteration our taboo list T of length $maxt = 6$ looks as follows:*

$$T = ((3, 1), (4, 9), (3, 9), (7, 2), (6, 7), (11, 3)).$$

Now assume that our TS has found a better solution that requires making the move $v = (4, 5)$. We then get a new T by doing

$$T := T \oplus (\bar{v} = (5, 4)) = ((4, 9), (3, 9), (7, 2), (6, 7), (11, 3), (5, 4)).$$

We now have the appropriate data structure and operations in place to begin considering which moves to choose when selecting a new neighbourhood and hereby choosing which one to add to our taboo list T.

1.10. Neighbourhood Search

Recall from section 1.8 that in JSSP we denote the current processing order with π, the associated set of moves by $V(\pi)$, the neighbourhood belonging to π by $H(\pi)$, and let C^* be the best solution value so far achieved (i.e. in JSSP the shortest makespan so far). We will now divide the possible moves in $V(\pi)$ into 3 categories:

- *Unforbidden (U-moves):* these comprise the moves from the set $V(\pi)\backslash T$.
- *Forbidden but profitable (FP-moves):* the forbidden moves in general comprise the moves from the set $V(\pi) \cap T$. A subset of these may actually yield a shorter makespan, although they are forbidden. The FP-moves therefore consist of the set

$$A = \{v \in V(\pi) \cap T \mid C_{max}(Q(\pi, v)) < C^*\}.$$

- *Forbidden and non-profitable (FN-moves):* these are then made up of the set $(V(\pi) \cap T)\backslash A$.

Normally in TS one would always choose a U-move or an FN-move, among these of course choosing the one that yields the best solution (i.e. the shortest makespan). We now turn back to defining which strategy to use if no U- or FP-moves are available. The best strategy for selecting among FN-moves in previous articles has been less important (selection has been made at random), but in Nowicki and Smutnicki ([3]) the neighbourhoods are a lot smaller and therefore the problem of choosing the best move among FN-moves becomes more important, as the situation in which only FN-moves are left to choose from is more often occurring, i.e. the situation in which the set $V(\pi)\backslash T$ is empty.

In [3] the authors propose a revised strategy of choosing the "oldest" move in our taboo list T, as well as making certain modifications to the list while doing so. They split the situation in two cases:

1. The possible moves $V(\pi)$ contains only 1 move. This move is then trivially selected.
2. The possible moves $V(\pi)$ contains more than 1 move. In this case we "fill" our taboo list from the right with the element in position $maxt$ (i.e. we perform $T :=$

$T \oplus T_{maxt}$), until $V(\pi)\backslash T \neq \emptyset$. At some point after performing a maximum of $maxt$ insertions, the set $V(\pi)\backslash T$ will contain only 1 single move which is then chosen as the next move for the TS iteration.

Example 6. *Recall our taboo list of length $maxt = 6$ from Example 5:*

$$T = ((3,1),(4,9),(3,9),(7,2),(6,7),(11,3)).$$

Suppose now that at some iteration we are left with a set of possible moves consisting only of FN-moves:

$$V(\pi) = \{(6,7),(3,9),(11,3)\}.$$

We now have to choose one of these moves using the above stated algorithm. We append the move $(11,3)$ to T, until we get only one element in $V(\pi)\backslash T$:

$$\begin{array}{ll} T = ((4,9),(3,9),(7,2),(6,7),(11,3),(11,3)) & V(\pi)\backslash T = \emptyset \\ T = ((3,9),(7,2),(6,7),(11,3),(11,3),(11,3)) & V(\pi)\backslash T = \emptyset \\ T = ((7,2),(6,7),(11,3),(11,3),(11,3),(11,3)) & V(\pi)\backslash T = \{(3,9)\} \end{array}$$

We then choose $v = (3,9)$ as our next move in the algorithm and add $\bar{v} = (9,3)$ to the taboo list:

$$T = ((6,7),(11,3),(11,3),(11,3),(11,3),(9,3))$$

The above discussion can be formalised into a more detailed and implementable Neighbourhood Search Procedure (NSP). The procedure takes as input the current processing order π, a non-empty set of possible moves $V(\pi)$, a taboo list T, and the so far shortest known makespan C^*. In the end it returns the next move to be performed (v'), the new processing order inferred by the move v' (π') and the revised taboo list (T').

Algorithm 1 NSP(π, $V(\pi)$, T, C^*)

```
1:  $A = \{v \in V(\pi) \cap T \mid C_{max}(Q(\pi, v)) < C^*\}$ // Find all FP-moves
2:  if $((V(\pi)\backslash T) \cup A \neq \emptyset)$ then
3:    // There are U- or FP-moves available. Select the best one:
4:    Select $v' \in (V(\pi)\backslash T) \cup A$ such that
      $C_{max}(Q(\pi, v')) = \min\{C_{max}(Q(\pi, v)) \mid v \in (V(\pi)\backslash T) \cup A\}$.
5:  else
6:    // There are only FN-moves available:
7:    if $(|V(\pi)| = 1)$ then
8:      Select $v' \in V(\pi)$.
9:    else
10:     while $(V(\pi)\backslash T = \emptyset)$ do
11:       $T \leftarrow T \oplus T_{maxt}$
12:     Select $v' \in V(\pi)\backslash T$.
13: $\pi' \leftarrow Q(\pi, v')$
14: $T' \leftarrow T \oplus \bar{v'}$
15: return $(v', \pi', T')$
```

1.11. The TS Algorithm

Using the NSP algorithm from the above section it is now a fairly easy task to design a complete algorithm for taboo search in JSSP. The Taboo Search Algorithm (TSA) takes as input an initial solution (processing order) π^* and creates an initially empty taboo list T. The general flow of the algorithm consist in first finding the moves for the initial solution: $V(\pi^*)$. We then choose a move using the NSP algorithm previously stated, which selects move $v' \in V(\pi^*)$ giving us a new neighbour $\pi := Q(\pi^*, v')$ to use in the next iteration, as well as an added entry to our taboo list ($T := T \oplus \bar{v}'$).

There are some other implementation details that we have omitted here, but a more thorough implementation is given in the algorithm TSA below. Omitted details include the number of iterations allowed (*maxiter*, fixed), the current iteration (t), and the length of the taboo list (*maxt*, fixed).

Algorithm 2 TSA(π^*)

```
1: $C^* \leftarrow C_{max}(\pi^*)$
2: $\pi \leftarrow \pi^*$
3: $T \leftarrow \{\}$
4: $t \leftarrow 0$
5: while ($t < maxiter$) do
6:   // Continue to iterate until reaching maxiter iterations:
7:   $t \leftarrow t + 1$
8:   Find $V(\pi)$
9:   if ($V(\pi) = \emptyset$) then
10:    // Stop. $\pi$ is optimal.
11:    return ($\pi, C^*$)
12:  else
13:    // Use NSP to find the next neighbour and update relevant values:
14:    ($v', \pi', T$) $\leftarrow$ NSP ($\pi, V(\pi), T, C^*$)
15:  if ($C_{max}(\pi') < C^*$) then
16:    $\pi \leftarrow \pi'$
17:    $C^* = C_{max}(\pi)$
18:    $t \leftarrow 0$
19: return ($\pi, C^*$)
```

The TSA algorithm is initially usable for taboo searching in JSSP. There are, however, some considerations that may have a great influence on the performance of the algorithm in general. Obviously, the values of *maxiter* and *maxt* will vary from problem to problem. In [3] the authors argue that the primal solution C^* only has a weak influence on the result of the algorithm. A bad initial solution only leads to a few extra iterations, not an overall worse result in terms of minimum makespan. The authors also present an improved algorithm (TSAB) that takes into account a form of long-term memory, helping the algorithm resume search at previously unvisited neighbourhoods of good solutions generated so far. The scope of this note, however, does not allow us to go into further details with this refinement.

1.12. A Few Remarks on Computational Results

We shall not delve very deeply into the computational results of the TSA and (more relevant) the TSAB algorithm. We will, however, mention that in 1996 the algorithm outperformed most other JSSP algorithms in various parameters (CPU time, closest to optimal makespan, etc.). The authors also suggested initial values for the parameters used in the algorithm, which can have an effect on the running time of the algorithm depending on the problem at hand. Without investigating matters we believe, though, that the TSAB algorithm must be outdated by today's standards. After all, a period of 12 years has passed since the release of the article. However, today the concepts of the article are still heavily in use as fundamental building blocks of more complex and refined algorithms for solving JSSP instances.

1.13. Conclusion

We have presented an overview of JSSP in the context that we wish to describe. In doing so we have established adequate detailed notation as well as re-introducing notions of blocks and critical paths. Also, previous neighbourhood-defining strategies have been listed and compared to a more appropriate definition that allows our problem to be solved more efficiently. In solving our problem we have introduced the taboo search paradigm and seen how it can be applied to JSSP in conjunction with the revised definition of a neighbourhood to provide faster solutions.

References

[1] P. Brucker, B. Jurisch, B. Sievers: *A branch and bound algorithm for the job-shop scheduling problem*, in *Discrete Applied Mathematics 49*, p107-127 (1994).

[2] P.J.M. van Laarhoven, E.H.L. Arts, J.K. Lenstra: *Job Shop Scheduling by Simulated Annealing*, in *Operations Research, Vol. 40, No. 1*, p113-125 (1992).

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

A. Appendix

A.1. List of variables

J	Set of jobs.
M	Set of machines.
O	Set of operations.
l_j	Number of operations in jobs 1 through j.
o_i	Number of operations in job i.
μ_i	The i'th machine, $i \in M$.
τ_i	Processing time of the i'th operation.
S_i	Starting time of the the i'th operation.
M_k	Set of all operations having $\mu_i = k$
$m_k = \|M_k\|$	Number of operations to be executed on machine k.
π_k	Some permutation of the elements in M_k.
Π_k	Set of all permutations of M_k or all possible instances of π_k.
R	Edges between operations of the same job. Known as conjunctive edges in [1].
$E(\pi)$	Edges between operations on the same machine using the permutation π. Known as disjunctive edges in [1].
u	Critical path in the graph G.
$C_{max}(\pi)$	Makespan using processing order π.