Using Tabu Search to Solve the Job Shop Scheduling Problem with Sequence Dependent Setup Times

Keith Schmidt (kas@cs.brown.edu)

May 18, 2001

1 Abstract

In this paper, I discuss the implementation of a robust tabu search algorithm for the Job Shop Scheduling problem and its extension to efficiently handle a broader class of problems, specifically Job Shop instances modeled with sequence dependent setup times.

2 Introduction

Motivation The Job Shop Scheduling problem is among the NP-Hard [6] problems with the most practical usefulness. Industrial tasks ranging from assembling cars to scheduling airplane maintenance crews are easily modeled as instances of this problem, and improving solutions by even as little as one percent can have a significant financial impact. Furthermore, this problem is interesting from a theoretical standpoint as one of the most difficult NP-Hard problems to solve in practice. To cite the canonical example, one 10x10 (that is, 10 jobs with 10 operations each) instance of this problem – denoted MT10 in the literature – was introduced by Muth and Thompson in 1963, but not provably optimally solved until 1989.

Definition The Job Shop Scheduling problem is formalized as a set J of n jobs, and a set M of m machines. Each job J_i has n_i subtasks (called *operations*), and each operation J_{ij} must be scheduled on a predetermined machine, $\mu_{ij} \in M$ for a fixed amount of time, d_{ij}, without interruption. No machine may process more than one operation at a time, and each operation $J_{ij} \in J_i$ must complete before the next operation in that job ($J_{i(j+1)}$) begins. The successor of operation x on its job is denoted $SJ[x]$, and the successor of x on its machine is denoted $SM[x]$. Likewise, the predecessors are denoted $PJ[x]$ and $PM[x]$. Every operation

x has a *release* (start) time denoted r_x, and *tail* time denoted t_x which is the the longest path from the time x is completed to the end.

Sequence dependent setup times Sequence dependent setup times are a tool for modeling a problem where there are different "classes" of operations which require machines to be reconfigured. For example two tasks in a machine shop may both be performed on the same drill press, but require different drill bits. In an instance of the Job Shop Scheduling problem with sequence dependent setup times, we assign a class identifier c_{ij} to each operation and we impose a fixed setup cost $p_{c_{ij},c_{i'j'}}$ to scheduling an operation of class $c_{i'j'}$ immediately after an operation of class c_{ij} on the same machine.

Objective functions To solve this problem, we must, for each machine, find an ordering of the operations to be scheduled on it that optimizes the objective function. There are several objective functions which are frequently applied to this problem. Far and away the most common is the minimization of the *makespan*, or the total time to complete all tasks. This objective function is widely used because it models many industrial problems well, and because it is very easy to compute efficiently. Others of note are the minimization of the total (weighted) tardiness, which is useful when modeling a problem where each job has its own due date, and minimization of total (weighted) cost, which is useful for modeling problems in which there is a cost associated with the operation of a machine.

Overview of local search techniques Since the first local search algorithms were tailored for the job shop problem in late 1980's, many different approaches have been developed.

P.J.M. van Laarhoven et al.[13] introduced the first simulated annealing algorithm for the job shop problem in 1988. That same year, H. Matsuo et al. [9] introduced a similar algorithm which was considerably more efficient. Since then, there has been considerable technical improvement, and an algorithm of Aarts et al. [1] published in 1994 is now the standard bearer of the area in terms of mean percentage error from the optimal [14]. Genetic Algorithms have also flourished as an area of study. Yamada and Nakano[15] introduced one of the first such algorithms tailored to this problem in 1992. Two years later, Aarts et al. [1] published one that was fairly efficient and robust. In 1995, Della Croce, et al.[5] presented another good algorithm amongst a flurry of activity. Tabu Search has also been an active field of study. Taillard [12] introduced the first tabu search-based algorithm in 1989. Dell'Amico and Trubian [4] pushed forward with several new advances in 1993. Barnes and Chambers [3] unveiled another tabu search algorithm in 1995, and Nowicki and Smutnicki [10] published a

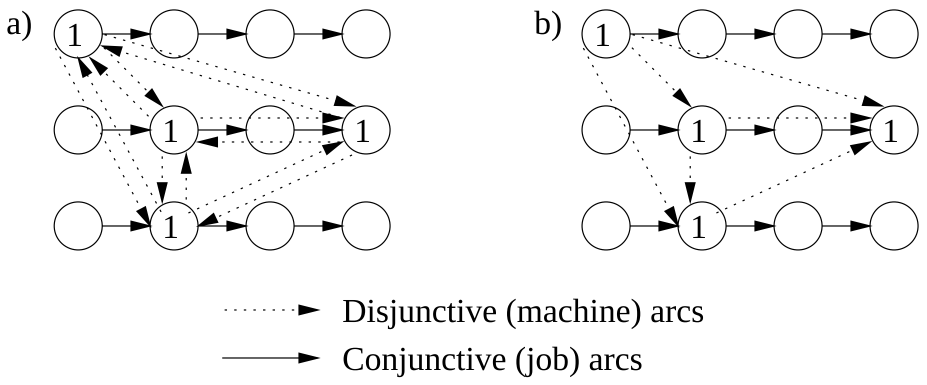

Figure 1: An illustration of a disjunctive graph

fast and robust one in 1996. One other entry of note is the “Guided Local Search” algorithm of Balas and Vazacopoulos [2] first published in 1994. While slow, it tends to find very good solutions on hard instances of the job shop problem.

Vaessens, et al. [14] in their 1996 survey of local search algorithms demonstrate that the available tabu search algorithms dominate the genetic algorithms and perform substantially better than the simulated annealing algorithms in most cases. The guided local search algorithms of Balas and Vazacopoulos compares favorably with the robust tabu search algorithms on the data sets tested. Hence tabu search seems to be a good basis framework for exploring a wider class of problems.

Representations Over the decades of research into solving the job shop scheduling problem with relative computational efficiency, several different ways to represent the problem have been introduced.

Disjunctive Graph The disjunctive graph representation for scheduling problems was first introduced by Roy and Sussmann in 1964 [11]. In this representation, the problem is modeled as a directed graph with the vertices in the graph representing operations, and with edges representing precedence constraints between operations. More precisely, a directed edge (v_1, v_2) exists if the operation at v_1 completes before the operation at v_2 begins.

These edges are divided into two sets called *conjunctive arcs* and *disjunctive arcs*. The conjunctive arcs are the precedences deriving from the ordering of the operations on their respective jobs. These edges are inherent in the problem definition and exist irrespective of the machine configurations. The disjunctive arcs, on the other hand, represent the precedence constraints imposed by the machine orderings. Before an ordering is imposed, $\forall x_i, y_i$ to be performed on machine M_i, there exist two conjunctive arcs, (x_i, y_i) and (y_i, x_i). Selecting a machine ordering is performed by removing exactly one arc from each pair to form a directed acyclic subgraph.

In figure 1 is an example of a subset of a disjunctive graph where 4 operations are to be scheduled on machine 1. In diagram **a** none of machine 1's disjunctive arcs have been selected, and so every operation has a pair of disjunctive arcs linking it with every other operation on the same machine. In diagram **b** is a selection of disjunctive arcs which defines an ordering of the operations on machine 1.

Earliest Start / Latest Completion Times The earliest start time of an operation, and its corresponding latest completion time are necessary to produce a usable schedule. The earliest start time (or release time) of an operation x is defined as the longest path from the start of the problem to x. Since the disjunctive graph must be acyclic to be a valid schedule, the release times will all be finite. The latest start time is almost the symmetric case. Computing the longest path from an operation x to the end of a problem will produce the tail time for that operation. The latest starting time of x is then equal to $makespan - t_x$. The release and tail times can be computed in linear time. This is done in a constructive manner: since $r_x = MAX(r_{PJ[x]} + d_{PJ[x]}, r_{PM[x]} + d_{PM[x]})$ the values of $r_{PJ[x]}$ and $r_{PM[x]}$ can be computed, stored, and used to compute r_x. In this case, we visit each edge in a schedule a constant number of times, all of the release and tail times can be computed in time linear in the number of operations.

Critical Path A *critical path* of a solution s to an instance of the job shop scheduling problem is a list of operations which determines the length of time s takes to complete. In other words, the length of the critical path is equal to the value of the makespan. In a disjunctive graph representation, the critical path is the longest path in the solution graph. When defined in terms of earliest start (ES) and latest end (LE) times, the following properties hold for all operations on the critical path:

$$ES_{x_0} = 0 \qquad LE_{x_i} = ES_{x_{i+1}} \qquad ES_{x_i} + d_{x_i} = LE_{x_i}$$

3 Exploring the neighborhood

Overview The performance of a local search algorithm, both in terms of the quality of solutions, and in the time required to reach them is heavily dependent on the neighborhood

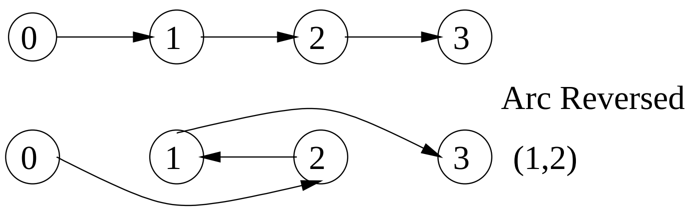

Figure 2: An illustration of the neighborhood N1

structure. Formally, given a solution s a *neighborhood* is a set $N(s)$ of candidate solutions which are adjacent to s. This means that if we are currently examining solution s the next solution we examine will be some $s' \in N(s)$. Typically, the solutions in $N(s)$ are generated from s with small, local modifications to s commonly called *moves*.

A neighborhood function must strike a balance between efficient exploration and wide coverage of the solution space. Using neighborhoods which are small and easy to evaluate may not allow the program to find solutions very different from the initial solution, while using those that are very large may take a long time to converge to a reasonably good solution. Some properties that seem to be useful for job shop neighborhood functions are described below, as are several neighborhood functions described in the literature.

Ideals The two overriding goals for designing neighborhoods for the job shop scheduling problem are *feasibility* and *connectivity*. A neighborhood with the former property ensures that, if provided a feasible solution, all neighboring solutions will be feasible as well. The latter ensures that there exists some finite sequence of moves between any feasible solution and a globally minimal solution.

Feasibility is important because, unlike some other combinatorial problems, infeasible configurations cannot be easily evaluated in a meaningful way (e.g. every infeasible configuration has a makespan of infinite length). Moreover, restoring feasibility from an infeasible configuration is, in general, a computationally expensive task which would dominate the time required to perform a move.

Connectivity is a desirable because it demonstrates that a globally minimal solution is reachable; without it, a local search algorithm is implicitly abandoning the hope of finding an optimal solution. It should be noted that connectivity guarantees the existence of a path to an optimal solution from any point in the solution space, but it gives no assistance in constructing that path.

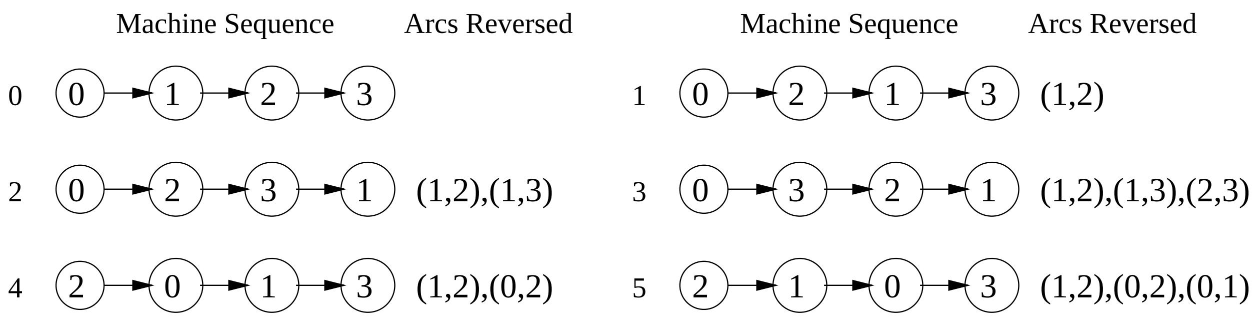

Figure 3: An illustration of the neighborhood NA

N1 and N2 The neighborhood now denoted N1 is a simple neighborhood concerning arcs which lie on a critical path. Specifically, given a solution s, every move leading to a solution in $N(s)$ reverses one machine arc on a critical path in s. Reversing an arc (x, $SM[x]$) consists of locally reordering the machine tasks [$PM[x]$, x, $SM[x]$, $SM[SM[x]]$] (figure 2) to form ($PM[x]$, $SM[x]$, x, $SM[SM[x]]$). N1 was first introduced by van Laarhoven [13] in 1988, in which paper it was demonstrated that N1 satisfied both the feasibility and connectivity criteria.

N2 is a neighborhood derived from N1 which reduces the number of neighboring solutions. N2 also reverses arcs on the critical path of a solution, but it does not consider an arc (x, $SM[x]$) if both ($PM[x]$, x) and ($SM[x]$, $SM[SM[x]]$) lie on a critical path, because the reversal of arc (x, $SM[x]$) cannot improve the makespan. This restriction is valid because, since there is no slack on the critical path, $r_{SM[SM[x]]} = r_{PM[x]} + d_x + d_{SM[x]}$. This is clearly independent of the orientation of the selected arc. Unfortunately, the reduction in the size of the neighborhood comes at a price – N2 does not preserve connectivity.

NA and RNA The neighborhoods NA and RNA were introduced by Dell'Amico and Trubian [4] in 1993. NA, like N1 concerns itself with arcs on the critical path of a solution. However, instead of examining one edge at a time, NA considers the permutation of up to 3 operations at a time. In figure 3, the operations $(0, 1, 2, 3)$ are assumed to all lie on a critical path in the problem. The primary arc being investigated is $(1, 2)$; in all of the the 5 modifications to this sequence, operation 2 precedes operation 1. In the first modified solution $(1, 2)$ is the only arc reversed. In the second, arc $(1, 2)$ is reversed, and then the arc $(1, 3)$ in the resultant intermediate solution is reversed. The other 3 permutations follow similarly. This neighborhood is obviously a superset of N1, so it preserves connectivity, and the authors prove that it preserves feasibility as well. RNA is a variant of NA restricted the

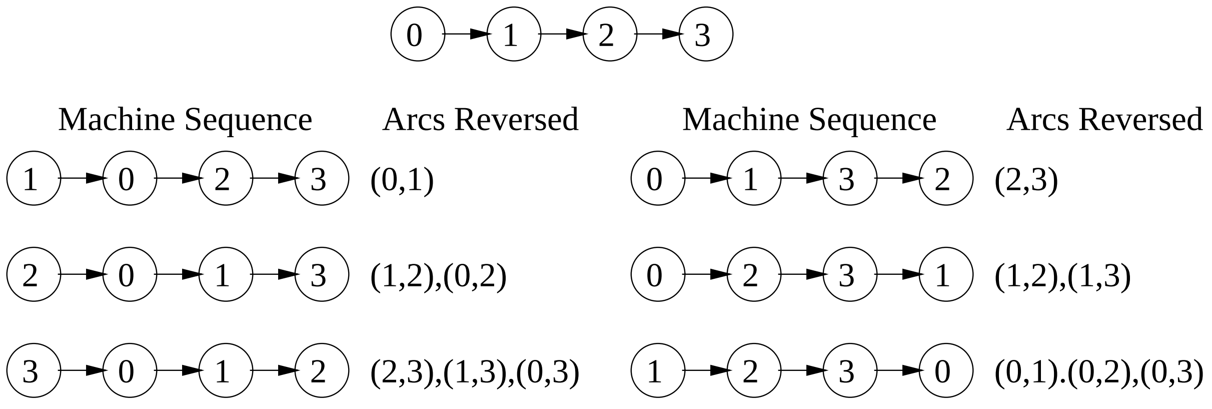

Figure 4: An illustration of the neighborhood NB

same way that N2 is: it does not consider an arc (x, $SM[x]$) if both ($PM[x]$, x) and ($SM[x]$, $SM[SM[x]]$) lie on a critical path.

NB the neighborhood NB was also introduced by Dell'Amico and Trubian [4] in 1993. NB operates on "blocks" of critical operations, defined as sets of consecutively scheduled operations on a single machine, all of which belong to a critical path. In this neighborhood, an operation is moved either toward the start or the end of its block. More specifically, an operation x in a block is swapped with its predecessor (or successor) as long as that swap produces a feasible configuration or until it is swapped with the first (or last) operation in that block. The original authors proved the connectivity of NB. In figure 4, it is assumed that the original sequence $[0, 1, 2, 3]$ is a block of critical path operations. The permutations in the left column all swap one of the operations in the block to the front of the block. The permutations on the right swap one of the block's operations to the end.

This neighborhood has the potential to swap a considerable number of arcs in one move, and as a result, it is not guaranteed to preserve feasibility. Hence, it becomes necessary to test for feasibility before each swap. Performing an exact feasibility test would require $O(nm)$ time and would severely affect the running time of this neighborhood as the number of swaps required for each block b is $O(b^2)$. To circumvent this, a constant time – but inexact – test is proposed. To wit, operation x is not scheduled before operation y if $r_{SJ[y]} + d_{SJ[y]} \leq r_{PJ[x]}$ because this indicates the possibility of an existing path from y to x.

4 General tabu search framework

Tabu Search is a meta-heuristic for guided local search which deterministically tries to avoid recently visited solutions. Specifically, the algorithm maintains a *tabu list* of moves which are forbidden. The list follows a FIFO rule and is typically very short (i.e. the length is frequently $O(\sqrt{N})$, where N is the total number of operations in the instance). Every time a move is taken, that move is placed on the tabu list.

The neighborhood The neighborhood function is the most important part of the tabu search algorithm, as it significantly affects both the running time and the quality of solutions. The neighborhood used in this implementation is one introduced by Dell'Amico and Trubian [4], which they call NC. NC is the union of the neighborhoods RNA and NB. NC is connected because NB is, and NC is a smaller neighborhood than NA because each arc examined in NC leads to fewer than 5 possible adjacent moves.

The tabu list The items placed on the tabu list are the reversed arcs, and a move is considered tabu if any of its component arcs are tabu. This model is used because, in the case of neighborhoods which may reverse multiple arcs, making only the move itself tabu would allow many substantively similar moves (i.e. those which share arcs with the tabu move) to be taken.

5 Generating an initial solution

List Scheduling There has been a great deal of research to find good, efficient heuristics to the job shop scheduling problem. Notably among these are the so-called *List Scheduling* (or *Priority Dispatch*) algorithms. These are constructive heuristics which examine a subset of operations and schedule these operations one at a time. While there are no guarantees on their quality, these algorithm have the advantage of running in sub-quadratic time (in normal use), and producing reasonable result with any of a number of good priority rules. List scheduling algorithms were first developed in the mid 1950's, and until about 1988 were the only known techniques for solving arbitrary large (≥ 100 element) instances.

While List Scheduling algorithms are no longer considered to be the state of the art for solving large job shop instances, they can still produce good initial solutions for local search algorithms. One of the most popular is the *Jackson Schedule* which selects the operation with the most work remaining (i.e. with the greatest tail time).

```
TabuSearch($JSSP$)
 1  ▷ $JSSP$ is an instance of the Job Shop Scheduling problem
 2  $sol \leftarrow$ InitialSolution($JSSP$)
 3  $bestCost \leftarrow$ cost($sol$)
 4  $bestSolution \leftarrow sol$
 5  $tabuList \leftarrow \emptyset$
 6  while keepSearching()
 7  do $N_{valid}(sol) \leftarrow \{s \in N(sol) | Move[sol, s] \notin tabuList\}$
 8     if $N_{valid}(sol) \neq \emptyset$
 9        then $sol' \leftarrow x \in N_{valid}(sol) | \forall y \in N_{valid}(sol)$ cost($x$) $\leq$ cost($y$)
10     updateTabuList($sol'$)
11     if cost($Move[sol, sol']$) $< bestCost$
12        then $bestSolution \leftarrow sol'$
13             $bestCost \leftarrow$ cost($sol'$)
14     $sol \leftarrow sol'$
15  return $bestSolution$
```

Figure 5: Pseudocode for a tabu search framework

```
List-Schedule($JSSP$)
 1  ▷ $JSSP$ is an instance of the Job Shop Scheduling problem
 2  ▷ $L$ is a list, $t$ is an operation, $\mu_t$ is the machine on which $t$ must run
 3  for each $Job\ J_i \in JSSP$
 4  do $L \leftarrow L \cup first[J_i]$
 5  for each $Machine\ M_i \in JSSP$
 6  do $avail[M_i] \leftarrow 0$;
 7  while $L \neq \emptyset$
 8  do $t \leftarrow$ bestOperation($L$)
 9     $\mu_t[avail[\mu_t]] \leftarrow t$
10     $avail[\mu_t] \leftarrow avail[\mu_t] + 1$
11     $L \leftarrow L \setminus t$
12     if $t \neq last[J_t]$
13        then $L \leftarrow L \cup$ jobNext($t$)
```

Figure 6: Pseudocode for a List Scheduling algorithm

Bidirectional List Scheduling Bidirectional List Scheduling[4] is an extension of the basic list scheduling framework. In this algorithm, one starts with two lists; one initialized with the first operation of each job and the other with the last operation of each job. The algorithm then alternates between lists, scheduling one operation and updating any necessary data each time, until all operations are scheduled. This algorithm aims to avoid a critical problem with basic list scheduling algorithms, namely that as they near completion, most of the operations are scheduled poorly (with respect to their priority rule) because the better placements have already been taken.

Additionally, the proposed bidirectional search chooses from the respective lists using a *cardinality-based semi-greedy heuristic with parameter* c[7], which means that the priority rule selects an operation uniformly at random from amongst the c operations with the lowest priority. This provides for a greater diversity of initial solutions which means that over several successive runs, a local search algorithm will explore a larger amount of total solution space than would otherwise be possible. In this implementation, the parameter c was set to 3.

6 Tweaking the tabu search

The tabu search framework described in figure 5 shows the tabu search in its canonical form. In practice, several modifications are made to this framework to improve the quality of solutions found, and to reduce the amount of time spent on computation. There are two high-level goals for improving the quality of solutions. The first is to attempt to visit nearby improving solutions that would be unreachable. The second goal is to increase the total amount of the solution space the tabu search visits. The former tries to ensure that all nearby local optima are explored to find reasonable solutions quickly. The latter tries to find solutions close to a global optimum by visiting many different areas of the solution space.

6.1 Efficiency

Fast Estimation One optimization critical to an efficient local search algorithm is the rapid computation of the value of a neighboring solution. Ideally it is possible to perform an exact evaluation quickly, but if this cannot be done, a good estimation will suffice. In the present problem, computing the exact value of the makespan for a neighboring solution is expensive. However, we can find the value of a reasonable estimation in time proportional to the number of arcs reversed by the move. That is, we can compute the value of the longest path through the affected arcs. To recompute the release times of an affected node x, we

```
Bidirectional-List-Schedule($JSSP$)
 1  ▷ $S$ and $T$ are lists of unscheduled operations, $L$ and $R$ are sets of scheduled operations
 2  $N \leftarrow \sum_{J_i} n_i$ ▷ N is the number of operations in JSSP
 3  for each $Job\ J_i \in JSSP$
 4  do $S \leftarrow S \cup first[J_i]$
 5      $T \leftarrow T \cup last[J_i]$
 6  $\forall x \in S\ r_x \leftarrow 0 \qquad \forall x \in T\ t_x \leftarrow 0$
 7  $L \leftarrow \emptyset \qquad R \leftarrow \emptyset$
 8  for each $Machine\ M_i \in JSSP$
 9  do $firstAvail[M_i] \leftarrow 0$
10      $lastAvail[M_i] \leftarrow |M_i| - 1$
11  ▷ Priority Rule: choose $s \in S$ $(t \in T)$ such that the longest known path through $s$ $(t)$ is minimal
12  while $|R| + |L| < N$
13  do for each $s \in S$
14      do ▷ $t'_x$ is the tail time of $x$ considering only already scheduled operations
15          $est[s] \leftarrow r_s + d_s + \text{MAX}(d_{SJ[s]} + t'_{SJ[s]}, \text{MAX}(d_x + t'_x)|x \in \mu_s, x \text{ is unscheduled })$
16      $choice \leftarrow S$[SemiGreedy-With-Parameter-c($est, c$)]
17      swap($\mu_{choice}[firstAvail[\mu_{choice}]], choice$)
18      $firstAvail[\mu_{choice}] \leftarrow firstAvail[\mu_{choice}] + 1$
19      $S \leftarrow S \setminus choice \quad L \leftarrow L \cup choice$
20      if $choice \in T$
21         then $T \leftarrow T \setminus choice$
22      if $SJ[choice] \notin R$
23         then $S \leftarrow S \cup SJ[choice]$
24       ▷ recompute the release times of the operations in S
25       ▷ recompute the tail times of the unscheduled operations to set up for step 2
26      if $|L| + |R| < N$
27         then for each $t \in T$
28                 do ▷ $r'_x$ is the release time of $x$ considering only already scheduled operations
29                     $est[s] \leftarrow \text{MAX}(d_{SJ[s]} + r'_{SJ[s]}, \text{MAX}(d_x + r'_x)|x \in \mu_s, x \text{ is unscheduled }) + d_s + t_s$
30                 $choice \leftarrow T$[SemiGreedy-With-Parameter-c($est, c$)]
31                 swap($\mu_{choice}[lastAvail[\mu_{choice}]], choice$)
32                 $lastAvail[\mu_{choice}] \leftarrow lastAvail[\mu_{choice}] - 1$
33                 $T \leftarrow T \setminus choice \quad R \leftarrow R \cup choice$
34                 if $choice \in S$
35                    then $S \leftarrow S \setminus choice$
36                 if $PJ[choice] \notin L$
37                    then $T \leftarrow T \cup PJ[choice]$
38                  ▷ recompute the tail times of the operations in T
39                  ▷ recompute the release times of the unscheduled operations to set up for step 1
40  ▷ recompute all release and tail times in fully scheduled JSSP
```

Figure 7: Pseudocode for a Bidirectional List Scheduling algorithm

```
SemiGreedy-With-Parameter-c($L, c$)
 1  ▷ $L$ is a list, $c$ is an integer
 2  ▷ $cLowestElements$ is a list of $c$ elements of $L$ which are the smallest seen to date
 3  ▷ $cLowestOrderStatistics$ is a list of the ranks of the elements in $cLowestElements$
 4  ▷ rand() is a function which returns a number uniformly at random from the interval [0, 1)
 5  if $size[L] < c$
 6    then return $\lfloor size[L] \cdot \textsc{rand}() \rfloor$
 7    else  for $i \leftarrow 1$ to $c$
 8            do $cLowestElements[i] = \infty$
 9          for $i \leftarrow 1$ to $size[L]$
10            do for $j \leftarrow 1$ to $c$
11                 do if $L[i] < cLowestElements[j]$
12                      then break
13               for $k \leftarrow c - 1$ to $j + 1$
14               do $cLowestElements[k] = cLowestElements[k - 1]$
15                  $cLowestOrderStatistics[k] = cLowestOrderStatistics[k - 1]$
16               if $j < c$
17                 then $cLowestElements[j] = L[i]$
18                      $cLowestOrderStatistics[j] = i$
19  return $cLowestOrderStatistics[\lfloor c \cdot \textsc{rand}() \rfloor]$
```

Figure 8: Pseudocode for an implementation of a cardinality-based semi-greedy heuristic with parameter c

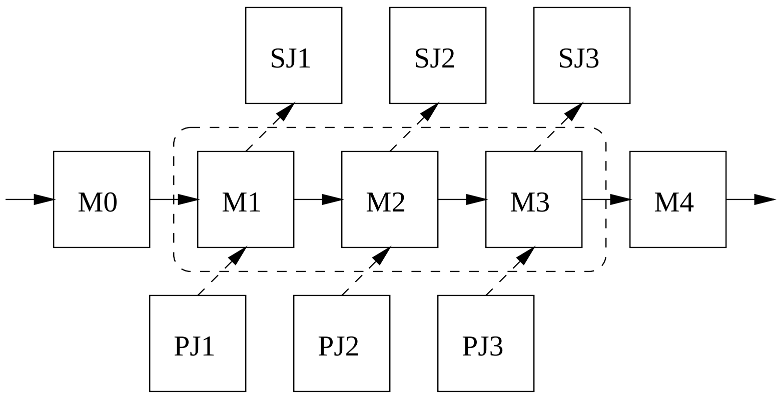

Figure 9: A portion of a schedule with a newly resequenced machine

need only to consider $PM[x]$ and $PJ[x]$; likewise, to recompute the tail times, we only need to examine x's two successors. The proof of this is fairly straightforward. A node x_0's release time can only be changed by modifying a node x_1 if x_1 lies on some path from the start to x_0. Since the nodes modified succeed their predecessors, the release times of their predecessors remain unchanged. A symmetric argument gives us the same result for the tail times of the successors.

Consider the example in figure 9. The set of operations $\{M1, M2, M3\}$ have just been re-sequenced on their machine. The release time of $M1$, in the new schedule,r'_{M1}, is $MAX(r_{M0}+d_{M0}, r_{PJ1}+d_{PJ1})$, the new release time of $M2$ is $MAX(r'_{M1}+d_{M0}, r_{PJ2}+d_{PJ2})$, and so forth.

Tabu list implementation Another optimization important to the overall running time of a Tabu search algorithm is the implementation of the Tabu list. While it is convenient to think of this structure as an actual list, in practice, implementing it as such results in a significant amount of computational overhead for all but the smallest lists.

Another approach is to store a matrix of all possible operation pairs (i.e. arcs). A time stamp is affixed to an arc when it is introduced into the problem by taking a move, and the timestamping value is incremented after every move. With this representation, a tabu list query may be performed in constant time (i.e. $currTime - timeStamp_{ij} < length[tabuList]$). Furthermore, the tabu list may be dynamically resized in constant time.

6.2 Finding better solutions

The goal of any optimization algorithm is to quickly find (near-) optimal solutions. Tabu search has been shown to be well-suited to this task, but researchers have determined several conditions where it could perform better and have proposed techniques for overcoming these. The first concerns cases where the algorithm misses improving solutions in its own neighborhood, and the second concerns cases in which the algorithm spends much of its time examining unprofitable solutions (i.e. ones which will not lead to improving solutions).

Aspiration Criterion The *aspiration criterion* is a function which determines when it is acceptable to ignore the tabu-state of a move. The intent of this is to avoid bypassing moves which lead to substantially better solutions simply because those moves are currently marked as tabu. Conventionally, the aspiration criterion accepts an otherwise tabu move if the cost (or estimated cost) of the solution it leads to is better than the cost of the best solution discovered so far.

Resizing the tabu list Another technique, which complements the aspiration criterion involves modifying the length of the tabu list. Typically, the tabu list is shortened when better solutions are discovered, and lengthened when moves leading to worse solutions are taken. The main assumption behind this is that when a good solution is found, there may be more within a few moves. Increasing the number of valid neighboring moves makes finding these better solutions more likely. In this implementation, when the current solution is better than the previous one, the tabu list is shortened by 1 move (until min) and when the current solution is worse than the previous one, the tabu list is lengthened by one move (until max). As a special case, when a new overall best solution is found, the length of the tabu list is set to 1. min is selected uniformly at random from the interval $[2, 2 + \lfloor\frac{n+m}{3}\rfloor]$, and max is selected uniformly at random from the interval $[min + 6, min + 6 + \lfloor\frac{n+m}{3}\rfloor]$. min and max are reset every 60 iterations.

Restoring the Best Known Solution One way to avoid spending excessive amounts of time examining unprofitable solutions is to periodically reset the current solution to be the best known known solution. While this artificially narrows the total solution coverage of the algorithm, it does so in a manner designed to continually explore regions where good solutions have been found. The time to wait before resetting must be set very carefully. If the reset delay chosen is too short, the tabu search may not be able to escape local minima; if it is too long, much time is still wasted exploring poor solutions. In practice, with a reasonable delay time (e.g. 800 - 1000 iterations), resetting the current solution seems to improve the quality of solutions found while preserving low running times. In this implementation, the solution was reset every 800 iterations.

6.3 Expanded Coverage

It is important for a tabu search algorithm to cover as much of the solution space as possible to increase the probability of finding a better solution. One particular problem to overcome is cycling amongst solutions. Visiting the same solutions repeatedly wastes moves that could otherwise be leading the search to unexplored solutions. The tabu list prevents the algorithm from spinning in small, tight cycles by making recently visited solutions tabu. However, this cannot guard against cycles whose length is longer than the tabu list. There are two techniques which help alleviate this problem.

Cycle Avoidance The easier to implement (and less effective) approach is to adjust the length of the tabu list from time to time. The rationale behind this is that when the list is longer, it prevents longer cycles. However, it will also prevent moves which are not part of the cycle and which could potentially lead to unexplored areas of the solution space. The second approach is to select a *representative arc* for every move taken, and store a small amount of the solution state (e.g. the cost of the current solution) with it. The next time a move with this representative arc is examined, the stored state is compared with the current state. If the two agree, this demonstrates the possibility of being within a cycle. If too many consecutive moves meet this criteria, it is assumed that the search is in a cycle, and all such potentially cyclic moves are avoided in the next step. In this implementation, the representative arc was chosen to be the first arc reversed, and the maximum number of potentially cyclic moves allowed was set to 3.

Exhaustion of Neighboring Solutions Another problem arises when the tabu search algorithm has explored enough of the local area to make all neighboring moves tabu. If this is the case, and there are no neighboring moves which satisfy the aspiration criterion, the tabu search should terminate prematurely. The strategy used to avoid this is to pick a move at random from $N(s)$ and follow it. This provides some chance of escaping a well-examined area and moving toward unexplored solutions.

7 Results

Data Collected The results in figure 10 demonstrate the robustness of this approach on conventional benchmark instances for the Job Shop problem (i.e. without setup times). The data for each instance was gathered over 20 runs of the algorithm. The times recorded are the average time over 20 runs. In the cases where the algorithm's best solution was the known optimal solution, the multiplicity of its occurrence is indicated in parentheses. All runs were performed on a 440MHz Sun Ultra 10 workstation.

Stability of the algorithm As can be seen from figures 11, 12, 13, and 14, the overall quality of solutions changes slightly when small modifications are made to the algorithm. This can be measured by the *relative error* of a solution, which is the percentage by which the best solution in a run exceeds the optimal (or best known) solution. The mean relative error of the solutions in figure 10 is 0.57% Figure 11, shows the results of running a variant of TS using the unrestricted version of neighborhood NA along with NB. While it produces similar

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6	58	70.3	(20)55	55.0	55	4.0
MT10	1051	1171.7	935	944.5	930	8.7
MT20	1316	1431.4	(15)1165	1166.8	1165	16.4
ABZ5	1343	1424.1	1236	1238.8	1234	7.8
ABZ6	1043	1097.9	(7)943	944.4	943	8.2
ABZ7	743	807.0	669	677.8	656	20.7
ABZ8	792	826.6	674	686.6	(645-669)	23.1
ABZ9	817	852.0	699	707.6	(661-679)	20.3
ORB1	1230	1352.3	1064	1089.9	1059	9.2
ORB2	975	1107.5	(2)888	890.3	888	7.8
ORB3	1293	1389.7	1008	1030.4	1005	9.3
ORB4	1118	1212.5	(1)1005	1015.2	1005	8.5
ORB5	1037	1167.6	889	897.4	887	8.1

Figure 10: Results for 20 runs of algorithm TS on job shop instances using neighborhood NC with initial solution from the bidirectional list scheduling algorithm

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6	58	66.8	(20)55	55.0	55	5.3
MT10	1018	1164.1	934	944.1	930	12.6
MT20	1349	1426.3	(2)1165	1176.3	1165	29.3
ABZ5	1313	1460.4	1236	1238.6	1234	9.8
ABZ6	979	1078.1	(20)943	943.0	943	9.3
ABZ7	767	810.8	672	686.1	656	31.2
ABZ8	781	833.5	679	692.5	(645-669)	30.2
ABZ9	792	858.5	703	720.9	(661-679)	29.7
ORB1	1173	1335.0	1060	1093.4	1059	13.3
ORB2	991	1097.5	889	893.0	888	9.8
ORB3	1243	1340.7	1015	1036.0	1005	12.9
ORB4	1108	1191.9	1011	1019.2	1005	12.3
ORB5	1068	1200.4	891	897.9	887	10.8

Figure 11: Results for 20 runs of algorithm TS on job shop instances, neighborhood NA $\cup$ NB with initial solution from the bidirectional list scheduling algorithm

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6	66	66.4	(20)55	55.0	55	4.0
MT10	1413	1416.4	937	947.4	930	9.1
MT20	1960	1968.0	1178	1214.7	1165	31.7
ABZ5	1463	1498.6	1236	1239.9	1234	10.0
ABZ6	1200	1217.0	(20)943	943.0	943	9.5
ABZ7	916	928.3	668	679.6	656	20.9
ABZ8	1078	1091.0	680	690.8	(645-669)	20.9
ABZ9	1063	1073.8	697	707.4	(661-679)	20.1
ORB1	1648	1669.7	(1)1059	1088.1	1059	13.1
ORB2	1253	1253.0	889	891.8	888	9.8
ORB3	2004	2004.0	1020	1039.8	1005	13.1
ORB4	1286	1286.0	1011	1016.9	1005	8.5
ORB5	1389	1443.6	889	894.7	887	8.2

Figure 12: Results for 20 runs of algorithm TS on job shop instances, neighborhood NC with initial solution from a list schedule with a Most-Work-Remaining priority rule

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6	57	68.5	(20)55	55.0	55	3.9
MT10	1076	1164.3	936	943.8	930	8.8
MT20	1361	1440.4	(16)1165	1166.0	1165	16.2
ABZ5	1313	1429.3	1236	1238.4	1234	7.6
ABZ6	1018	1100.7	(6)943	944.7	943	7.3
ABZ7	788	812.9	668	678.1	656	20.2
ABZ8	807	847.0	677	684.4	(645-669)	20.3
ABZ9	813	850.8	698	706.5	(661-679)	19.4
ORB1	1219	1346.6	1060	1085.7	1059	9.1
ORB2	1003	1100.8	889	892.4	888	7.5
ORB3	1249	1342.3	1020	1028.0	1005	9.2
ORB4	1116	1198.2	1011	1017.6	1005	8.5
ORB5	1047	1173.6	891	896.9	887	8.1

Figure 13: Results for 20 runs of algorithm TS on job shop instances, neighborhood NC with initial solution from the bidirectional list scheduling algorithm, without restoring the best known solution

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6	58	68.2	(20)55	55.0	55	4.0
MT10	1085	1163.5	(1)930	942.4	930	8.9
MT20	1319	1416.1	(14)1165	1167.0	1165	16.2
ABZ5	1351	1429.0	1236	1238.9	1234	7.9
ABZ6	1035	1104.5	(7)943	944.7	943	7.5
ABZ7	774	810.0	670	678.5	656	21.0
ABZ8	785	831.4	677	690.0	(645-669)	20.8
ABZ9	801	851.8	695	708.0	(661-679)	20.0
ORB1	1240	1364.4	1064	1087.8	1059	9.1
ORB2	993	1089.3	(1)888	890.4	888	7.7
ORB3	1256	1333.1	(1)1005	1035.2	1005	9.2
ORB4	1128	1209.7	(2)1005	1014.9	1005	8.5
ORB5	1049	1178.0	889	896.2	887	8.1

Figure 14: Results for 20 runs of algorithm TS on job shop instances using neighborhood NC with initial solution from the bidirectional list scheduling algorithm, without resetting the *min* and *max* bounds on the size of the tabu list

results for many of the problems, its mean relative error is 0.79%. Figure 12, shows the results of running a variant of TS whose starting solution is from a unidirectional list scheduling algorithm with a Most-Work-Remaining priority rule, and with 0.81% mean relative error. Figure 13 shows the results of running a variant of TS where the current solution is never reset to the best known solution; its mean relative error is 0.72%. Lastly, figure 14 shows the results of running a variant of TS where the bounds on the length of the tabu list are never reset. This gives slightly better results, with a mean relative error of 0.50%, even though the average final solution tends to be slightly worse than in the original TS. These results indicate that the algorithm TS is fairly well-tuned for instances of the job shop problem without setup times. In essence, this shows that TS should give good solutions to instances of the job shop scheduling problem with sequence dependent setup times, and indicates that better results may be had by modifying the algorithm.

8 Sequence Dependent Setup Times

The variant of job shop scheduling which includes sequence-dependent setup times shares a great deal of structure with the original. One important consequence is that job shop neighborhoods which are connected or maintain feasibility across moves preserve these properties when setup times are included. One notable difference lies in the suitability of re-

```
SETUPTIMEGENERATE($JSSP$)
1   ▷ rand() is a function which returns a number uniformly at random from the interval [0, 1)
2   $numClasses \leftarrow \frac{numOperations}{10}$
3   $maxTransitionCost \leftarrow \frac{\sum_{O_{ij}} d_{ij}}{numOperations}$
4   for each operation $O_{ij}$
5   do $class[O_{ij}] \leftarrow \lfloor \text{RAND}() \cdot numClasses \rfloor$
6   for each class $c_0$
7   do for each class $c_1$
8       do if $c_0 = c_1$
9           then $p_{c_0,c_1} \leftarrow 0$
10          else $p_{c_0,c_1} \leftarrow \lfloor \text{RAND}() \cdot maxTransitionCost \rfloor$
```

Figure 15: Pseudocode of sequence-dependent setup time instance generation

stricted neighborhoods. Recall that restricting N1 to N2 (which does not consider arcs internal to a block) was deemed valid because, since there is no slack on the critical path, $r_{SM[SM[x]]} = r_{PM[x]} + d_x + d_{SM[x]}$. However, when sequence dependent setup times are introduced, $r_{SM[SM[x]]} = r_{PM[x]} + p_{c_{PM[x]},c_x} + d_x + p_{c_x,c_{SM[x]}} + d_{SM[x]} + p_{c_{SM[x]},c_{SM[SM[x]]}}$ Furthermore, this restriction can only be valid if $p_{c_{PM[x]},c_x} + p_{c_x,c_{SM[x]}} + p_{c_{SM[x]},c_{SM[SM[x]]}} = p_{c_{PM[x]},c_{SM[x]}} + p_{c_{SM[x]},c_x} + p_{c_x,c_{SM[SM[x]]}}$, which is not true in general.

Data generation The instance data for problems with sequence dependent setup times were generated from existing job shop instances of varying difficulties (MT6, MT10, MT20, ABZ5, ABZ6, ABZ7, ABZ8, ABZ9). For each generated instance, the number of distinct classes was set to $\frac{numOperations}{10}$. Each operation was assigned a class selected uniformly at random from the available classes. The setup times for operations in the same class was set to 0, and all other setup times were integers selected uniformly at random from the interval $[0, \frac{\sum_{O_{ij}} d_{ij}}{numOperations})$. (see fig. 15 for the implementation).

8.1 Results

Data Collected In figures 16, 17 and 18 are the computational results for the job shop instances with sequence dependent setup times. Figure 16 displays the results for 20 runs of this tabu search algorithm using neighborhood NC, and figure 17 shows the results of the runs on the same data sets, but using the neighborhood (NA ∪ NB). The lower bounds on the optimal solution are the best known lower bounds for the corresponding problems without transition times. The upper bounds are the best results obtained from several long

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6-TT	62	69.3	(12)55	55.4	55	4.2
MT10-TT	1177	1346.3	1037	1050.9	(930-1018)	8.7
MT20-TT	1592	1714.1	1322	1343.6	(1165-1316)	15.7
ABZ5-TT	1534	1669.9	1333	1359.2	(1234-1325)	7.7
ABZ6-TT	1122	1230.3	1002	1027.1	(943-1002)	7.8
ABZ7-TT	889	950.9	760	771.6	(656-752)	20.9
ABZ8-TT	921	981.5	774	789.6	(645-772)	23.1
ABZ9-TT	958	1000.3	785	795.2	(661-776)	20.2

Figure 16: Results for 20 runs of algorithm TS on instances with sequence dependent setup times, using neighborhood NC and initial solution from Bidir

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6-TT	60	71.2	(5)55	55.8	55	5.5
MT10-TT	1199	1328.0	1026	1059.5	(930-1018)	12.9
MT20-TT	1631	1755.8	1328	1378.6	(1165-1316)	30.1
ABZ5-TT	1578	1651.9	1355	1370.6	(1234-1325)	10.5
ABZ6-TT	1163	1256.1	1009	1028.5	(943-1002)	10.3
ABZ7-TT	893	958.1	762	784.5	(656-752)	32.0
ABZ8-TT	927	968.5	788	800.6	(645-772)	30.7
ABZ9-TT	923	992.9	791	807.4	(661-776)	30.4

Figure 17: Results for 20 runs of algorithm TS on instances with sequence dependent setup times using neighborhood (NA $\cup$ NB) and initial solution from Bidir

Instance	Init. sol. (best)	Init. sol. (mean)	Final sol. (best)	Final sol. (mean)	Optimal Value	time (sec)
MT6-TT	66	66.5	(13)55	55.4	55	4.0
MT10-TT	1413	1423.2	1026	1052.3	(930-1018)	9.1
MT20-TT	1960	1962.4	1320	1347.2	(1165-1316)	15.9
ABZ5-TT	1463	1501.7	1335	1357.9	(1234-1325)	7.9
ABZ6-TT	1200	1212.8	1008	1028.7	(943-1002)	7.8
ABZ7-TT	916	928.8	758	768.1	(656-752)	20.8
ABZ8-TT	1078	1095.7	772	788.1	(645-772)	20.7
ABZ9-TT	1063	1081.2	778	790.5	(661-776)	19.9

Figure 18: Results for 20 runs of algorithm TS on instances with sequence dependent setup times using neighborhood NC and initial solution from a list schedule with a Most-Work-Remaining priority rule

runs of algorithm TS. Figure 18 displays the results for 20 runs of this tabu search algorithm starting from a unidirectional list schedule and using neighborhood NC.

Analysis of Variants The mean relative error of the basic TS algorithm is 0.62% (figure 16). In the variant where the unrestricted version of NA is used in conjunction with NB, the mean relative error is 1.25% (figure 17). Surprisingly, NC (RNA $\cup$ NB) provided slightly better results on average than (NA $\cup$ NB) even though the theoretical justification for the restriction of NA does not hold for these instances. Figure 18 reports the results of running a variant of TS where the initial solution is computed with a list scheduling algorithm using a Most-Work-Remaining priority rule; its mean relative error is 0.44%. This variant provided better overall solutions than the first algorithm even though the initial solutions were often poorer. This seems to indicate that finding a very good starting solution is not as important to instances with setup times as it is to instances without setup times.

9 Conclusions

This research has demonstrated that it is possible to take existing tabu search algorithms and adjust them to provide reasonable solutions to a wider class of problems. As is evident from the data, the initial solution provided by the bidirectional list scheduling algorithm is substantially poorer for the instances with sequence dependent setup times than for those instances without them. This is likely because the bidirectional list scheduling algorithm does nothing to prevent large setup times on the machine arcs connecting the left and right halves. Even so, the Bidirectional list schedule typically found better initial solutions that those found by the unidirectional list schedule tested. However, the neighborhood NC was able to converge to slightly better solutions when using the “poorer” initial starting solutions provided by the unidirectional list schedule.

Unfortunately, without further work on the instances with sequence dependent setup times, the relative error from the optimal values cannot be established accurately for most of them.

10 Future work

Among the questions that could be addressed in future research are:

- Is there a solid theoretical justification for restricted neighborhoods behaving better than their unrestricted counterparts on problem instances with sequence dependent setup times?

- Is it possible to reasonably extend these algorithms to even broader classes of job shop problems? (e.g. A wider class of objective functions).
- What are some other neighborhood functions which are better suited to solving problem instances with sequence dependent setup times?
- What are some other heuristics that are better suited to providing good initial solutions to problem instances with sequence dependent setup times?
- Where can a good source of data for problem instances arising in industry be found?

References

[1] E.H.L. Aarts, P.J.M. van Laarhoven, J.K. Lenstra, and N.L.J. Ulder, "A Computational Study of Local Search Algorithms for Job Shop Scheduling", *ORSA Journal on Computing 6*, (1994)118-125.

[2] E. Balas and A. Vazacopoulos, "Guided Local Search with Shifting Bottleneck for Job Shop Scheduling", *Management Science Research Report*, Graduate School of Industrial Administration, Carnegie Mellon University (1994).

[3] J.W. Barnes and J.B. Chambers, "Solving the Job Shop Scheduling Problem Using Tabu Search", *IIE Transactions 27*, (1994)257-263.

[4] M. Dell'Amico and M. Trubian, "Applying tabu search to the job-shop scheduling problem", *Annals of Operations Research*, 41(1993)231-252.

[5] F. Della Croce, R. Tadei, and G. Volta, "A Genetic Algorithm for the Job Shop Problem", *Computers and Operations Research*, 22(1995)15-24.

[6] M.R. Garey, D.S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling", *Mathematics of Operations Research*, 1(1976)117-129.

[7] J.P. Hart and A.W. Shogan, "Semi-greedy heuristics: an empirical study", *Operations Research Letters* 6(1987)107-114.

[8] A.S. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present, and future", *European Journal of Operational Research*, 113(1999)390-434.

[9] H. Matsuo, C.J. Suh, and R.S. Sullivan, “A Controlled Search Simulated Annealing Method for the General Jobshop Scheduling Problem”, *Working Paper 03-04-88*, Graduate School of Business, University of Texas, Austin.

[10] E. Nowicki and C. Smutnicki, “A Fast Taboo Search Algorithm for the Job Shop Problem”, *Management Science*, 6(1996)797-813.

[11] B. Roy and B. Sussmann, “Les problems d’ordonnancement avec constraintes disjonctives”, *Node DS n.9 bis*, SEMA, Montrouge (1964).

[12] E. Taillard, “Parallel Taboo Search Techniques for the Job Shop Scheduling Problem”, *ORSA Journal on Computing 6*, (1994)108-117.

[13] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra, “Job shop scheduling with simulated annealing”, *Report OS-R8809*, Centre for Mathematics and Computer Science, Amsterdam (1988).

[14] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra, “Job Shop Scheduling by Local Search”, *INFORMS Journal on Computing*, 3(1996)302-317.

[15] T. Yamada and R. Nakano, “A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems”, *Parallel Problem Solving from Nature 2*, R. Männer, B. Mandrick (eds.), North-Holland, Amsterdam, (1992)281-290.

A Code

A.1 DataStructures.H

```
/**
  * FILE: DataStructures.H
  * AUTHOR: kas
  * RAISON D'ETRE: data structures for modeling the shifting
  * bottleneck heuristic for the Job Shop Scheduling problem.
  */
#define NULL 0
#define FALSE 0
#define TRUE 1

#ifndef DATA_STRUCTURES_H
#define DATA_STRUCTURES_H

#include <iostream.h>
/* *******************************************************************
 *
 * CLASS: List
 *
 * *******************************************************************/

using namespace std;

template <class T>
class List {

public:

  class ListNode {

  public:
    ListNode() {                                                    ListNode
      data_ = NULL;
      next_ = NULL;
      prev_ = NULL;
    }

    virtual ~ListNode() {                                           ~ListNode
      if (next_)
        delete next_;
    }

    void setNext(const ListNode* const next) {                      setNext
      next_ = (ListNode*)next;
    }

    void setPrev(const ListNode* const prev) {                      setPrev
      prev_ = (ListNode*)prev;
    }
  void setData(const T data) {                                setData
    data_ = data;
  }

  const T data() const {                                      data
    return data_;
  }

  const ListNode* const next() const {                        next
    return next_;
  }

  const ListNode* const prev() const {                        prev
    return prev_;
  }

private:
  T data_;
  ListNode* next_;
  ListNode* prev_;
};

typedef ListNode Node;

List() {                                                      List
  headPtr_ = NULL;
  tailPtr_ = NULL;
  size_ = 0;
}

virtual ~List(){                                              ~List
  if (headPtr_)
    delete headPtr_;
}

void addFirst(T toAdd) {                                      addFirst
  Node* newNode = new Node();
  newNode->setData(toAdd);
  newNode->setNext(headPtr_);
  newNode->setPrev(NULL);
  if (tailPtr_ == NULL) {
    tailPtr_ = newNode;
  }
  else {
    headPtr_->setPrev(newNode);
  }
  headPtr_ = newNode;
  size_++;
}

void addLast(T toAdd) {                                       addLast
  Node* newNode = new Node();
  newNode->setData(toAdd);
  newNode->setPrev(tailPtr_);
  newNode->setNext(NULL);
  if (headPtr_ == NULL) {
    headPtr_ = newNode;
  }
  else {
    tailPtr_->setNext(newNode);
  }
  tailPtr_ = newNode;
  size_++;
}

void addAfter(T toAdd, Node* curr) {                                  addAfter

  Node* newNode = new Node();
  newNode->setData(toAdd);

  newNode->setNext(next(curr));
  newNode->setPrev(curr);
  curr->setNext(newNode);

  if (curr == tailPtr_) {
    tailPtr_ = newNode;
  }
  else {
    next(newNode)->setPrev(newNode);
  }
  size_++;
}

bool addAtIndex(T toAdd, unsigned int idx) {                          addAtIndex

  if (idx == 0) {
    addFirst(toAdd);
    size_++;
    return TRUE;
  }

  else if (idx == size_) {
    addLast(toAdd);
    size_++;
    return TRUE;
  }
  else if (idx > 0 && idx < size_) {
    int i = 0;
    Node* ptr = first();
    while (NULL != ptr) {
      if (i == (idx - 1)) {
        Node* newNode = new Node();
        newNode->setData(toAdd);

        newNode->setNext(next(ptr));
        newNode->setPrev(ptr);
        next(newNode)->setPrev(newNode);
        ptr->setNext(newNode);
        size_++;
        return TRUE;
      }
      ptr = next(ptr);
      i++;
    }
  }
  else { return FALSE; }
}

void removeItem(T toDelete) {                                        removeItem

  Node* f_ptr = first();

  while (NULL != f_ptr && f_ptr->data() != toDelete) {
    f_ptr = next(f_ptr);
  }
  if (f_ptr != NULL && f_ptr->data() == toDelete) {
    if (prev(f_ptr) != NULL) {
      prev(f_ptr)->setNext(next(f_ptr));
    }
    if (next(f_ptr) != NULL) {
      next(f_ptr)->setPrev(prev(f_ptr));
    }
    if (f_ptr == headPtr_) {
      headPtr_ = next(f_ptr);
    }
    if (f_ptr == tailPtr_) {
      tailPtr_ = prev(f_ptr);
    }
    f_ptr->setNext(NULL);
    f_ptr->setPrev(NULL);
    f_ptr->setData(NULL);
    delete f_ptr;
    size_--;
  }
}


Node* findItem(T toFind) const {                                     findItem
  Node* ptr = first();
  while (NULL != ptr) {
    if (ptr->data() == toFind)
      return ptr;
    ptr = next(ptr);
  }
  return NULL;
}

int findIndex(T toFind) const {                                      findIndex
  int i = 0;
  Node* ptr = first();
  while (NULL != ptr) {
    if (ptr->data() == toFind)
      return i;
    ptr = next(ptr);
    i++;
  }
  return -1;
}

Node* first() const {                                  first
  return headPtr_;
}

Node* last() const {                                   last
  return tailPtr_;
}

void removeFirst() {                                   removeFirst
  if (headPtr_ != NULL) {
    Node* n = next(first());

    if (n != NULL) {
      n->setPrev(NULL);
    }

    headPtr_->setNext(NULL);
    headPtr_->setData(NULL);
    delete headPtr_;
    size_--;
    if (headPtr_ == tailPtr_) {
      tailPtr_ = n;
    }
    headPtr_ = n;
  }
}

void removeLast() {                                    removeLast
  if (tailPtr_ != NULL) {
    Node* p = prev(last());

    if (p != NULL) {
      p->setNext(NULL);
    }

    tailPtr_->setPrev(NULL);
    tailPtr_->setData(NULL);
    delete tailPtr_;
    size_--;
    if (headPtr_ == tailPtr_) {
      headPtr_ = p;
    }
    tailPtr_ = p;
  }
}
```

```
  Node* atRank(int rank) const {                                  atRank
    if (rank < 0 || rank >= size()) {
      return NULL;
    }
    else {
      Node* iter = first();
      for (int i = 0; i < rank; i++) {
        iter = next(iter);
      }
      return iter;
    }
  }

  Node* next(Node* curr) const {                                  next
    return (Node*)(curr->next());
  }

  Node* prev(Node* curr) const {                                  prev
    return (Node*)(curr->prev());
  }

  int size() const {                                              size
    return size_;
  }

private:

  int size_;
  Node* headPtr_;
  Node* tailPtr_;
};

/**
  * Some typedefs for cleaner code
  */

class Job;
class Operation;
class Machine;

typedef List<Job*> JobList;
typedef List<Operation*> OperationList;
typedef List<Machine*> MachineList;


/* ********************************************************************
 *
 * CLASS: Job
 *
 * ********************************************************************/

class Job {
```

```
public:

  Job();


  virtual ~Job();

  int numOperations() const;                                              numOperations
  void setNumOperations(int numOperations);

  void setAtRank(int i, Operation* toAdd);
  Operation* atRank(int i) const;

  Operation** operations() const;

  void dump() const;

private:

  int size_;
  Operation** operationVector_;

};


/* ******************************************************************
 *
 * CLASS: Machine
 *
 * ******************************************************************/
class Machine {

public:

  Machine();

  Machine(const OperationList* const opList);

  virtual ~Machine();

  int numOperations() const;                                              numOperat
  void setNumOperations(int numOperations);

  void setAtRank(int i, Operation* toAdd);
  Operation* atRank(int i) const;

  Operation** operations() const;

  void dump() const;

private:
  Operation** operationVector_;
  int size_;
};
```

```
/* *****************************************************************
 *
 * CLASS: Operation
 *
 * Add accessors/ mutators for machine & time. add job_
 *
 * *****************************************************************/

class Operation {

public:

  typedef enum {
    HEAD = 0,
    TAIL
  } CumulativeType;

  Operation();

  Operation(const int job, const int jobIdx,                          Operation
            const int host, const double time);

  virtual ~Operation();

  int job() const;
  int jobIdx() const;

  int machineIdx() const;
  void setMachineIdx(int newIdx);

  int machine() const;
  void setMachine(int newMachine);

  void setTime(double newTime);
  double time() const;

  double cumulativeTime(CumulativeType type) const;
  void setCumulativeTime(CumulativeType type, double newTime);

  int operationClass() const;
  void setOperationClass(int newClass);

  double transitionTime() const;
  void setTransitionTime(double newTime);

  void dump() const;

private:

  int host_;
  int job_;

  int hostIdx_;
  int jobIdx_;

  double time_;
  double transitionTime_;

  double timeToReturn_;

  double cumulativeTime_[2];

  int operationClass_;


};

#endif
```

A.2 DataStructures.C

```
/**
 * FILE: DataStructures.H
 * AUTHOR: kas
 * RAISON D'ETRE: data structures for modeling the Job Shop
 * Scheduling problem.
 */

#ifndef DATA_STRUCTURES_H
#include "DataStructures.H"
#endif


#include <math.h>
#include <iostream.h>

/* *****************************************************************
 *
 * CLASS: Job
 *
 * Note: add Operation insertion.
 *
 * *****************************************************************/

Job::Job() {                                                          Job::Job
  size_ = 0;
  operationVector_ = NULL;
}


Job::~Job() {                                                         Job::~Job
  if (operationVector_)
    delete [] operationVector_;
}

Operation**
Job::operations() const {                                             Job::operations
  return operationVector_;
}

int
Job::numOperations() const {                                          Job::numOpera
  return size_;
}

void
Job::setNumOperations(int numOperations) {                            Job::setNumOp
  size_ = numOperations;
  if (operationVector_) {
    delete [] operationVector_;
  }
  operationVector_ = new Operation*[numOperations];
  for (int i = 0; i < size_; i++) {
    operationVector_[i] = NULL;
  }
}

void
Job::setAtRank(int i, Operation* toAdd) {                              Job::setAtRan
  if (i >= 0 && i < size_) {
    operationVector_[i] = toAdd;
  }
}

Operation*
Job::atRank(int i) const {                                             Job::atRank
  if (i >= 0 && i < size_) {
    return operationVector_[i];
  }
  else return NULL;
}

void
Job::dump() const {                                                    Job::dump

  if (operationVector_ == NULL) {
    cout << "Empty" << endl;
  }
  else {
    cout << "[ ";
    for (int i = 0; i < size_; i++) {
      if (operationVector_[i] != NULL){
        (operationVector_[i])->dump();
      }
      cout << endl;
    }
    cout << "]" << endl;
  }
}

/* *********************************************************************
 *
 * CLASS: Machine
 *
 * Note: add Operation insertion.
 *
 * *********************************************************************/

Machine::Machine() {                                                   Machine::Mach
  size_ = 0;
  operationVector_ = NULL;
}


Machine::~Machine() {                                                  Machine::~Mac
  if (operationVector_)
    delete [] operationVector_;
```

```
}

Operation**
Machine::operations() const {                                    Machine::op
  return operationVector_;
}

int
Machine::numOperations() const {                                 Machine::num(
  return size_;
}

void
Machine::setNumOperations(int numOperations) {                   Machine::se
  size_ = numOperations;
  if (operationVector_) {
    delete [] operationVector_;
  }
  operationVector_ = new Operation*[numOperations];
  for (int i = 0; i < size_; i++) {
    operationVector_[i] = NULL;
  }
}

void
Machine::setAtRank(int i, Operation* toAdd) {                    Machine::setAt
  if (i >= 0 && i < size_) {
    operationVector_[i] = toAdd;
  }
}

Operation*
Machine::atRank(int i) const {                                   Machine::atRa
  if (i >= 0 && i < size_) {
    return operationVector_[i];
  }
  else return NULL;
}

void
Machine::dump() const {                                          Machine::dum

  if (operationVector_ == NULL) {
    cout << "Empty" << endl;
  }
  else {
    cout << "( ";
    for (int i = 0; i < size_; i++) {
      if (operationVector_[i] != NULL){
        (operationVector_[i])->dump();
      }
      cout << endl;
    }
    cout << ")" << endl;
  }
}


/* ******************************************************************
 *
 * CLASS: Operation
 *
 * ******************************************************************/

Operation::Operation() {
  job_ = -1;
  host_ = -1;

  jobIdx_ = -1;
  hostIdx_ = -1;

  time_ = 0;
  cumulativeTime_[0] = 0;
  cumulativeTime_[1] = 0;
}

Operation::Operation(const int job, const int jobIdx,
                     const int host, const double time) {
  host_ = host;
  job_ = job;
  jobIdx_ = jobIdx;

  transitionTime_ = 0.0;
  time_ = time;

  operationClass_ = 0;
}

Operation::~Operation() {
}


int
Operation::job() const {
  return job_;
}

int
Operation::jobIdx() const {
  return jobIdx_;
}

int
Operation::machineIdx() const {
  return hostIdx_;
}

void
```

```
Operation::setMachineIdx(int newIdx) {
  hostIdx_ = newIdx;
}

int
Operation::machine() const {
  return host_;
}

void
Operation::setMachine(int newMachine) {
  host_ = newMachine;
}

double
Operation::time() const {
  return time_;
}

void
Operation::setTime(double newTime) {
  time_ = newTime;
}

double
Operation::cumulativeTime(CumulativeType type) const {
  return cumulativeTime_[(int)type];
}

void
Operation::setCumulativeTime(CumulativeType type, double newTime) {
  cumulativeTime_[(int)type] = newTime;
}

int
Operation::operationClass() const {
  return operationClass_;
}

void
Operation::setOperationClass(int newClass) {
  operationClass_ = newClass;
}

double
Operation::transitionTime() const {
  return transitionTime_;
}

void
Operation::setTransitionTime(double newTime) {
  transitionTime_ = newTime;
}
```

```
void
Operation::dump() const {                                                                    270 Operation::
  cout << "<M:(" << host_ << ", " << hostIdx_<< "), J:(" << job_
       << ", " << jobIdx_ << "), t:" << time_ << ", r,q:("
       << cumulativeTime_[0] << ", " << cumulativeTime_[1] <<")>";
}
```

A.3 TS_Solution.H

```
#include "DataStructures.H"

class Job;
class Machine;
class Operation;
class TabuList;
class CycleWitness;

class TS_Solution {


public:

  TS_Solution::TS_Solution(Job** jLists, Machine** mLists,                   TS_Solution::T
                           double** classTransitions,
                           int numJobs, int numMachines, int numClasses);

  virtual ~TS_Solution();

  const OperationList* const computeCriticalPath(Operation::CumulativeType type);

  void longestPathHelper(Operation* toCompute, Operation::CumulativeType);

  void longestPathHelperIncomplete(Operation* toCompute,
                                   Operation::CumulativeType type,
                                   const int* const lastFreeL,
                                   const int* const firstFreeR);

  void longestPathLinear(Operation::CumulativeType);

  Operation* jobPrev(const Operation* const curr) const;

  Operation* jobNext(const Operation* const curr) const;

  Operation* machinePrev(const Operation* const curr) const;

  Operation* machineNext(const Operation* const curr) const;

  Job* jList(int idx) const;
  Machine* mList(int idx) const;


  void swap(Operation* o1, Operation* o2);

  OperationList* criticalPath() const;

  int numJobs() const;
  int numMachines() const;

  double makespan() const;

  double transitionTime(int startClass, int endClass);
  TabuList* tabuList() const;

  CycleWitness* witness() const;

  void dump() const;

private:
  Job** jLists_;
  Machine** mLists_;

  int numJobs_;
  int numMachines_;

  double makespan_;

  OperationList* criticalPath_;

  TabuList* tabu_;

  CycleWitness* witness_;

  double** transitionMatrix_;
};


class TabuList {

private:
  // underlying data structure

  typedef struct {
    int endIdx_; // index of the end Node of the swap
    int timeStamp_;
  } TLData;

  int** tlMatrix_;


  int time_;
  int tlLength_;

  int numJobs_;
  int numOperations_;

public:

  // we expect each job to have the same number of operations.

  TabuList(int numJobs, int numOperations);

  virtual ~TabuList();

  bool query(const Operation* const start, const Operation* const end) const;        query
  void incrementTime();

  int currentTime() const;
  void reset(); // resets the time to 0 and cleans out the list.

  void updateLength(int newLength);

  int length() const;

  void mark(const Operation* const start, const Operation* const end);

};

class CycleWitness {

private:
  // underlying data structure

  typedef struct {
    int endIdx_; // index of the end Node of the swap
    double value_;
  } CWData;

  double** cwMatrix_;

  int numJobs_;
  int numOperations_;

  int cycleDepth_;
  int timeToBreak_;

public:

  // we expect each job to have the same number of operations.

  CycleWitness(int numJobs, int numOperations);

  virtual ~CycleWitness();

  bool query(const Operation* const start, const Operation* const end, int value) const;        query

  void mark(const Operation* const start, const Operation* const end, int value);

  void setTimeToBreak(int newTime);

  void adjustCycleDepth(bool queryVal);

  bool isInCycle() const;

  void reset(); // cleans out the list.
};
```

A.4 TS_Solution.C

```
#include "TS_Solution.H"

#define MAX(a,b) (((a) < (b)) ? (b) : (a))
/* ******************************************************************
 *
 * TS Solution
 *
 * ******************************************************************/

TS_Solution::TS_Solution(Job** jLists, Machine** mLists,                    TS_Solution
                         double** classTransitions,
                         int numJobs, int numMachines, int numClasses) {

  jLists_ = jLists;
  mLists_ = mLists;
  numJobs_ = numJobs;
  numMachines_ = numMachines;
  criticalPath_ = new OperationList();

  transitionMatrix_ = classTransitions;

  tabu_ = new TabuList(numJobs, jList(0)->numOperations());
  witness_ = new CycleWitness(numJobs, jList(0)->numOperations());
}

TS_Solution::~TS_Solution() {                                               TS_Solution::~

  delete tabu_;
  delete witness_;
  delete criticalPath_;
}

const OperationList* const
TS_Solution::computeCriticalPath(Operation::CumulativeType type) {         computeCritica

  int i, j;
  Operation* nextInPath;

  // clear out existing critical path.

  while (criticalPath_->first() != NULL) {
    criticalPath_->removeFirst();
  }

  if (type == Operation::HEAD) {
    // start with the end of the machines.
    for (i = 0; i < numJobs_; i++) {
      if (jList(i)->atRank(jList(i)->numOperations() -1)->cumulativeTime(type) +
          jList(i)->atRank(jList(i)->numOperations() -1)->time() == makespan_) {
        // we found the endpt of a critical path.
        nextInPath = jList(i)->atRank(jList(i)->numOperations() -1);
        break;
      }
    }
    while (jobPrev(nextInPath) != NULL || machinePrev(nextInPath) != NULL) {
      criticalPath_−>addFirst(nextInPath);
      if (jobPrev(nextInPath) != NULL && machinePrev(nextInPath) != NULL) {
        if (jobPrev(nextInPath)−>cumulativeTime(type) ==
            nextInPath−>cumulativeTime(type) − jobPrev(nextInPath)−>time()) {
          nextInPath = jobPrev(nextInPath);
        }
        else {
          nextInPath = machinePrev(nextInPath);
        }
      }
      else if (jobPrev(nextInPath) != NULL) {
        nextInPath = jobPrev(nextInPath);
      }
      else if (machinePrev(nextInPath) != NULL) {
        nextInPath = machinePrev(nextInPath);
      }
    }
    criticalPath_−>addFirst(nextInPath);
  }
  else { // type == Operation::TAIL
    // preserve the order of the critical path...

    // start with the end of the machines.
    for (i = 0; i < numJobs_; i++) {
      if (jList(i)−>atRank(0)−>cumulativeTime(type) + jList(i)−>atRank(0)−>time() == makespan_) {
        // we found the endpt of a critical path.
        nextInPath = jList(i)−>atRank(0);
        break;
      }
    }
    while (jobNext(nextInPath) != NULL || machineNext(nextInPath) != NULL) {

      criticalPath_−>addLast(nextInPath);
      if (jobNext(nextInPath) != NULL && machineNext(nextInPath) != NULL) {
        if (jobNext(nextInPath)−>cumulativeTime(type) ==
            nextInPath−>cumulativeTime(type) − jobNext(nextInPath)−>time()) {
          nextInPath = jobNext(nextInPath);
        }
        else {
          nextInPath = machineNext(nextInPath);
        }
      }
      else if (jobNext(nextInPath) != NULL) {
        nextInPath = jobNext(nextInPath);
      }
      else if (machineNext(nextInPath) != NULL) {
        nextInPath = machineNext(nextInPath);
      }
    }
    criticalPath_−>addLast(nextInPath);
  }

  return criticalPath_;
}

void
TS_Solution::longestPathHelper(Operation* toCompute, Operation::CumulativeType type) {        TS_Solution::lo

  if (toCompute−>cumulativeTime(type) > −HUGE_VAL) {
    return;
  }
  else {
    Operation* nextMachine;
    Operation* nextJob;

    double lj = 0, lm = 0, cumulative = 0;

    if (type == Operation::TAIL) {

      nextJob = jobNext(toCompute);
      if (nextJob != NULL) {
        longestPathHelper(nextJob, type);
        lj = nextJob−>cumulativeTime(type) + nextJob−>time();
      }

      nextMachine = machineNext(toCompute);
      if (nextMachine != NULL) {
        longestPathHelper(nextMachine, type);
        lm = (nextMachine−>cumulativeTime(type) + nextMachine−>time() +
              toCompute−>transitionTime());
      }
    }
    else {

      nextJob = jobPrev(toCompute);
      if (nextJob != NULL) {
        longestPathHelper(nextJob, type);
        lj = nextJob−>cumulativeTime(type) + nextJob−>time();
      }

      nextMachine = machinePrev(toCompute);
      if (nextMachine != NULL) {
        longestPathHelper(nextMachine, type);
        lm = (nextMachine−>cumulativeTime(type) + nextMachine−>time() +
              nextMachine−>transitionTime());
      }
    }

    cumulative = MAX(lj, lm);
    toCompute−>setCumulativeTime(type, cumulative);
  } // longest path not yet cached

}
```

```
// this will be used when we need to estimate the longest path of a
// partially scheduled machine. This is only necessary for generating
// an initial solution. I need to be able to determine if a given
// operation is unscheduled, and follow edges from that operation to
// the next operation on that machine which has been scheduled.

// suggestion: take the arrays indicating which machine operations
// have been scheduled. If the current Operation has not been
// scheduled, test the first Operation that has been scheduled.

void
TS_Solution::longestPathHelperIncomplete(Operation* toCompute, Operation::CumulativeType type,        longestPathHel
                                         const int* const lastFreeL, const int* const firstFreeR) {

  if (toCompute->cumulativeTime(type) > -HUGE_VAL) {
    return;
  }
  else {
    Operation* nextMachine;
    Operation* nextJob;
    double lj = 0, lm = 0, cumulative = 0;

    if (type == Operation::TAIL) {

      nextJob = jobNext(toCompute);
      if (nextJob != NULL) {
        longestPathHelperIncomplete(nextJob, type, lastFreeL, firstFreeR);
        lj = nextJob->time() + nextJob->cumulativeTime(type);
      }

      if (toCompute->machineIdx() >= lastFreeL[toCompute->machine()] &&
          toCompute->machineIdx() <= firstFreeR[toCompute->machine()]) {

        nextMachine = mList(toCompute->machine())->atRank(firstFreeR[toCompute->machine()] + 1);
        if (nextMachine != NULL) {
          longestPathHelperIncomplete(nextMachine, type, lastFreeL, firstFreeR);
          lm = (nextMachine->time() + nextMachine->cumulativeTime(type) +
                toCompute->transitionTime());
        }
      }

    }
    else {

      nextJob = jobPrev(toCompute);
      if (nextJob != NULL) {
        longestPathHelperIncomplete(nextJob, type, lastFreeL, firstFreeR);
        lj = nextJob->time() + nextJob->cumulativeTime(type);
      }

      if (toCompute->machineIdx() >= lastFreeL[toCompute->machine()] &&
          toCompute->machineIdx() <= firstFreeR[toCompute->machine()]) {

        nextMachine = mList(toCompute->machine())->atRank(lastFreeL[toCompute->machine()] - 1);
        if (nextMachine != NULL) {
          longestPathHelperIncomplete(nextMachine, type, lastFreeL, firstFreeR);
          lm = (nextMachine->time() + nextMachine->cumulativeTime(type) +
                  nextMachine->transitionTime());
        }
      }

    }
    cumulative = MAX(lj, lm);
    toCompute->setCumulativeTime(type, cumulative);
  } // longest path not yet cached

}


void
TS_Solution::longestPathLinear(Operation::CumulativeType type) {
```

TS_Solution::lo

```
  int i,j;
  OperationList rootSet;

  // can clean up the following code with abstractions. should do so...

  // initialize the values.
  if (type == Operation::TAIL) {
    for (i = 0; i < numJobs_; i++) {
      for (j = 0; j < jList(i)->numOperations(); j++) {
        Operation* curr = jList(i)->atRank(j);
        if (j == 0 &&
            curr->machineIdx() == 0) {
          // object is in initial set
          rootSet.addFirst(curr);
          curr->setCumulativeTime(type, −HUGE_VAL);
        }
        if (j == jList(i)->numOperations() − 1 &&
            curr->machineIdx() == mList(curr->machine())->numOperations() − 1) {
          // object is terminal
          curr->setCumulativeTime(type, 0.0);
        }
        else {
          curr->setCumulativeTime(type, −HUGE_VAL);
        }
      }
    }
  }
  else { // type == Operation::HEAD
    for (i = 0; i < numJobs_; i++) {
      for (j = 0; j < jList(i)->numOperations(); j++) {
        Operation* curr = jList(i)->atRank(j);
        if (j == 0 &&
            curr->machineIdx() == 0) {
          // object is terminal
          curr->setCumulativeTime(type, 0.0);
        }
        if (j == jList(i)->numOperations() - 1 &&
            curr->machineIdx() == mList(curr->machine())->numOperations() - 1) {
          // object is in initial set
          rootSet.addFirst(curr);
          curr->setCumulativeTime(type, -HUGE_VAL);
        }
        else {
          curr->setCumulativeTime(type, -HUGE_VAL);
        }
      }
    }
  }

  OperationList::Node* iter = rootSet.first();
  while (iter != NULL) {

    longestPathHelper(iter->data(), type);
    iter = rootSet.next(iter);
  }

  iter = rootSet.first();

  makespan_ = -HUGE_VAL;

  while (iter != NULL) {

    if (iter->data()->cumulativeTime(type) + iter->data()->time() > makespan_) {
      makespan_ = iter->data()->cumulativeTime(type) + iter->data()->time();
    }

    iter = rootSet.next(iter);
  }

}

Operation*
TS_Solution::jobPrev(const Operation* const curr) const {                        TS_Solution::jo
  if (curr->job() >=0 && curr->job() < numJobs_ &&
      curr->jobIdx() > 0 && curr->jobIdx() < jList(curr->job())->numOperations()) {
    return jList(curr->job())->atRank(curr->jobIdx() - 1);
  }
  else {
    return (Operation*)NULL;
  }
}

Operation*
TS_Solution::jobNext(const Operation* const curr) const {                        TS_Solution::jo
  if (curr->job() >=0 && curr->job() < numJobs_ &&
      curr->jobIdx() >= 0 && curr->jobIdx() < jList(curr->job())->numOperations() -1) {
    return jList(curr->job())->atRank(curr->jobIdx() + 1);
  }
  else {
    return (Operation*)NULL;
  }
}

Operation*
TS_Solution::machinePrev(const Operation* const curr) const {                    TS_Solution::m
  if (curr->machine() >=0 && curr->machine() < numJobs_ &&
      curr->machineIdx() > 0 && curr->machineIdx() < mList(curr->machine())->numOperations()) {
    return mList(curr->machine())->atRank(curr->machineIdx() - 1);
  }
  else {
    return (Operation*)NULL;
  }
}

Operation*
TS_Solution::machineNext(const Operation* const curr) const {                    TS_Solution::m
  if (curr->machine() >=0 && curr->machine() < numJobs_ &&
      curr->machineIdx() >= 0 &&
      curr->machineIdx() < mList(curr->machine())->numOperations() - 1) {
    return mList(curr->machine())->atRank(curr->machineIdx() + 1);
  }
  else {
    return (Operation*)NULL;
  }
}


Job*
TS_Solution::jList(int idx) const {                                              TS_Solution::jl
  return jLists_[idx];
}

Machine*
TS_Solution::mList(int idx) const {                                              TS_Solution::m
  return mLists_[idx];
}

void
TS_Solution::swap(Operation* o1, Operation* o2) {                                TS_Solution::s
  if (o1->machine() != o2->machine()) {
    return;
  }
  if (o1 == o2) {
    return;
  }

  int m = o1->machine();
  int tempIdx = o1->machineIdx();
  o1->setMachineIdx(o2->machineIdx());
  o2->setMachineIdx(tempIdx);
  mList(m)->setAtRank(o1->machineIdx(), o1);
  mList(m)->setAtRank(o2->machineIdx(), o2);

  // now to handle transition data
  int tempClass = o1->operationClass();
  o1->setOperationClass(o2->operationClass());
  o2->setOperationClass(tempClass);

  Operation* prev;
  Operation* next;

  if (o1->machineIdx() > 0) {
    prev = mList(m)->atRank(o1->machineIdx() - 1);
    prev->setTransitionTime(transitionTime(prev->operationClass(), o1->operationClass()));
  }
  if (o2->machineIdx() > 0) {
    prev = mList(m)->atRank(o2->machineIdx() - 1);
    prev->setTransitionTime(transitionTime(prev->operationClass(), o2->operationClass()));
  }

  if (o1->machineIdx() < mList(m)->numOperations() - 1) {
    next = mList(m)->atRank(o1->machineIdx() + 1);
    o1->setTransitionTime(transitionTime(o1->operationClass(), next->operationClass()));
  }
  if (o2->machineIdx() < mList(m)->numOperations() - 1) {
    next = mList(m)->atRank(o2->machineIdx() + 1);
    o2->setTransitionTime(transitionTime(o2->operationClass(), next->operationClass()));
  }
}

OperationList*
TS_Solution::criticalPath() const {                                    TS_Solution::c
  return criticalPath_;
}

int
TS_Solution::numJobs() const {                                         TS_Solution::n
  return numJobs_;
}

int
TS_Solution::numMachines() const {                                     TS_Solution::n
  return numMachines_;
}

double
TS_Solution::makespan() const {                                        TS_Solution::n
  return makespan_;
}

double
TS_Solution::transitionTime(int startClass, int endClass) {            TS_Solution::t
  return transitionMatrix_[startClass][endClass];
}

TabuList*
TS_Solution::tabuList() const {                                        TS_Solution::t
  return tabu_;
```

```
}

CycleWitness*
TS_Solution::witness() const {                                    TS_Solution::w
  return witness_;
}

void
TS_Solution::dump() const {                                       TS_Solution::d

  int i;

  cout << "Jobs " << endl;
  for (i = 0; i < numJobs_; i++) {
    jList(i)->dump();
    cout << endl;
  }

  cout << endl;
  cout << "Machines " << endl;
  for (i = 0; i < numMachines_; i++) {
    mList(i)->dump();
    cout << endl;
  }

}


/* *****************************************************************
 *
 * TABU LIST
 *
 * *****************************************************************/

TabuList::TabuList(int numJobs, int numOperations) {              TabuList::Tabu
  numJobs_ = numJobs;
  numOperations_ = numJobs*numOperations;

  time_ = 0;
  tlLength_ = 0;

  tlMatrix_ = new int*[numOperations_];
  for (int i = 0; i < numOperations_; i++) {
    tlMatrix_[i] = new int[numOperations_];
  }

  for (int i = 0; i < numOperations_; i++) {
    for (int j = 0; j < numOperations_; j++) {
      tlMatrix_[i][j] = -numOperations_;
    }
  }
}

TabuList::~TabuList() {                                           TabuList::~Tab
```

```
  for (int i = 0; i < numOperations_; i++) {
    delete [] tlMatrix_[i];
  }
  delete [] tlMatrix_;
}


// returns TRUE if a move is tabu, FALSE otherwise

bool
TabuList::query(const Operation* const start, const Operation* const end) const {

  int startIdx = numOperations_/numJobs_ * start->job() + start->jobIdx();
  int endIdx  = numOperations_/numJobs_ * end->job() + end->jobIdx();

  return tlMatrix_[startIdx][endIdx] + tlLength_ >= time_;
}

void
TabuList::incrementTime() {
  time_++;
}

int
TabuList::currentTime() const {
  return time_;
}

void
TabuList::reset() {
  time_ = 0;

  for (int i = 0; i < numOperations_; i++) {
    for (int j = 0; j < numOperations_; j++) {
      tlMatrix_[i][j] = -numOperations_;
    }
  }
}

void
TabuList::updateLength(int newLength) {
  tlLength_ = newLength;
}

int
TabuList::length() const {
  return tlLength_;
}

void
TabuList::mark(const Operation* const start, const Operation* const end) {
  int startIdx = numOperations_/numJobs_ * start->job() + start->jobIdx();
  int endIdx  = numOperations_/numJobs_ * end->job() + end->jobIdx();
  tlMatrix_[startIdx][endIdx] = time_;

}



/* ******************************************************************
 *
 * CYCLE WITNESS
 *
 * ******************************************************************/

CycleWitness::CycleWitness(int numJobs, int numOperations) {                CycleWitness::
  numJobs_ = numJobs;
  numOperations_ = numJobs*numOperations;

  cwMatrix_ = new double*[numOperations_];
  for (int i = 0; i < numOperations_; i++) {
    cwMatrix_[i] = new double[numOperations_];
  }

  for (int i = 0; i < numOperations_; i++) {
    memset(cwMatrix_[i], 0, numOperations_*sizeof(double));
  }
}

CycleWitness::~CycleWitness() {                                             CycleWitness::

  for (int i = 0; i < numOperations_; i++) {
    delete []cwMatrix_[i];
  }
  delete [] cwMatrix_;
}


// returns TRUE if an arc has the query value, FALSE otherwise

bool
CycleWitness::query(const Operation* const start, const Operation* const end, int value) const {   CycleWitness::

  int startIdx = numOperations_/numJobs_ * start->job() + start->jobIdx();
  int endIdx  = numOperations_/numJobs_ * end->job() + end->jobIdx();

  return cwMatrix_[startIdx][endIdx] == value;

}

void
CycleWitness::mark(const Operation* const start, const Operation* const end, int value) {   CycleWitness::
  int startIdx = numOperations_/numJobs_ * start->job() + start->jobIdx();
  int endIdx  = numOperations_/numJobs_ * end->job() + end->jobIdx();
  cwMatrix_[startIdx][endIdx] = value;

}

void
CycleWitness::reset() {

  for (int i = 0; i < numOperations_; i++) {
    memset(cwMatrix_[i], 0, numOperations_*sizeof(double));
  }
}

void
CycleWitness::setTimeToBreak(int newTime) {
  timeToBreak_ = newTime;
}

void
CycleWitness::adjustCycleDepth(bool queryVal) {
  if (queryVal) {
    cycleDepth_++;
  }
  else {
    cycleDepth_ = 0;
  }
}

bool
CycleWitness::isInCycle() const {
  return (cycleDepth_ > timeToBreak_);
}
```

A.5 Utilities.H

```
class Job;
class Machine;
class Operation;

void
parse(const char* fileName, int& numJobs, Job**& jobs, int& numMachines,
      Machine**& machines, int& numOps, Operation**& operations,
      int& numClasses, double**& classTranstions);
```

parse

A.6 Utilities.C

```
#include "DataStructures.H"

#include <iostream.h>
#include <fstream.h>
#include <string.h>

using namespace std;

void
parse(const char* fileName, int& numJobs, Job**& jobs, int& numMachines,            parse
      Machine**& machines, int& numOps, Operation**& operations,
      int& numClasses, double**& classTransitions) {

  int i, j;
  ifstream ifs(fileName);
  char buf[256];

  numClasses = 0;

  classTransitions = NULL;

  ifs >> buf;
  if (!strcmp(buf, "NUM_OPERATIONS")) {
    ifs >> numOps;
  }
  operations = new Operation*[numOps];

  ifs >> buf;
  if (!strcmp(buf, "NUM_JOBS")) {
    ifs >> numJobs;
    jobs = new Job*[numJobs];

    for (i = 0; i < numJobs; i++) {
      jobs[i] = new Job;
    }
  }
  else if (!strcmp(buf, "NUM_CLASSES")) {
    ifs >> numClasses;
    ifs >> buf;
    if (!strcmp(buf, "NUM_JOBS")) {
      ifs >> numJobs;
      jobs = new Job*[numJobs];

      for (i = 0; i < numJobs; i++) {
        jobs[i] = new Job;
      }
    }
  }

  ifs >> buf;
  if (!strcmp(buf, "NUM_MACHINES")) {
    ifs >> numMachines;
```

```
}
machines = new Machine*[numMachines];

for (i = 0; i < numMachines; i++) {
  machines[i] = new Machine;
}

int hostNum, numJobOps, numMachineOps, machineIdx;
double time;
int opClass;

numJobOps = numOps/numJobs;
numMachineOps = numOps/numMachines;

int* hostIdx = new int[numMachines];

for (int i = 0; i < numMachines; i++) {
  machines[i]->setNumOperations(numMachineOps);
  hostIdx[i] = 0;
}


for (int jobNum = 0; jobNum < numJobs; jobNum++) {
  jobs[jobNum]->setNumOperations(numJobOps);
  for (int jobIdx = 0; jobIdx < numJobOps; jobIdx++) {
    ifs >> hostNum;
    ifs >> time;
    operations[jobNum*numJobOps + jobIdx] =
      new Operation (jobNum, jobIdx, hostNum, time);

    if (numClasses > 0) {
      ifs >> opClass;
      operations[jobNum*numJobOps + jobIdx]->setOperationClass(opClass);
    }

    jobs[jobNum]->setAtRank(jobIdx, operations[jobNum*numJobOps + jobIdx]);
    operations[jobNum*numJobOps + jobIdx]->setMachineIdx(hostIdx[hostNum]);
    machines[hostNum]->setAtRank(hostIdx[hostNum], operations[jobNum*numJobOps + jobIdx]);
    hostIdx[hostNum]++;
  }
}

if (numClasses == 0) {
  numClasses = 1; // must do this for default table
}

classTransitions = new double*[numClasses];
for (i = 0; i < numClasses; i++) {
  classTransitions[i] = new double[numClasses];
}

if (numClasses > 1) {
  for (i = 0; i < numClasses; i++) {
    for (j = 0; j < numClasses; j++) {
        ifs >> classTransitions[i][j];
      }
    }
  }
  else {
    classTransitions[0][0] = 0.0;
  }

  delete [] hostIdx;
}
```

A.7 main.C

```
#include <alloca.h>
#include <unistd.h>
#include <sys/time.h>
#include <assert.h>

#include "TS_Solution.H"
#include "DataStructures.H"
#include "Utilities.H"

#define MAX(a,b) (((a) < (b)) ? (b) : (a))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))



typedef struct {
  int job_;
  int jobIdx_;
} OperationSig;

typedef struct {

  bool   hasMove_;
  double bestMove_;
  bool   moveIsNA_;

  int         naPermutation_;
  Operation* start_;
  Operation* end_;

  int       toMove_;
  int       destination_;
  int       toModify_;

} NeighboringSolutions;

/**
 * Prototypes
 **/

void tabuSearchJS(TS_Solution* ts);

double estimateLongestPath(Operation* sNode, Operation* eNode,
                           int& permutation, TS_Solution& tss);

double longestPath(const Machine* const m, int startIdx, int num, const TS_Solution& tss);

bool isOnCriticalPath(const Operation* const toTest, double makespan);

// Bidirectional list schedule
void initialSolution(TS_Solution* sol);

// Unidirectional list schedule with Most-Work-Remaining priority rule
```

estimateLonges

```
void initialSolution2(TS_Solution* sol);

int semiGreedy(int c, const double* const vals, int numVals);

void exploreNeighborhood(TS_Solution* sol);

void n1(NeighboringSolutions& nt, NeighboringSolutions& rand,
        TS_Solution* sol, double& num, double& reserveNum);
void n2(NeighboringSolutions& nt, NeighboringSolutions& rand,
        TS_Solution* sol, double& num, double& reserveNum);
void na(NeighboringSolutions& nt, NeighboringSolutions& rand,
        TS_Solution* sol, double& num, double& reserveNum);

void rna(NeighboringSolutions& nt, NeighboringSolutions& rand,
         TS_Solution* sol, double& num, double& reserveNum);

bool naMoveIsNotTabu(Operation* start, int permutation, TS_Solution* sol);

void
fillNASolutions(NeighboringSolutions& nt, NeighboringSolutions& rand,
                double currTest, int permutation, Operation* start, Operation* end,
                bool isCycle, bool isNotTabu, double& num, double& reserveNum);


double testNBMove(Machine* m, int toMove, int destination, TS_Solution* sol);
void nb(NeighboringSolutions& nt, NeighboringSolutions& rand,
        TS_Solution* sol, double& num, double& reserveNum);

void
fillNBSolutions(NeighboringSolutions& nt, NeighboringSolutions& rand,
                double currTest, int toMove, int destination, Machine* toModify,
                bool isCycle, bool isNotTabu, double& num, double& reserveNum);

bool keepSearching();
bool meetsAspirationCriterion(double estimate);

void print(const OperationList* const);

const bool RESET_SOLUTIONS = TRUE;
const int INITIAL_TABU_LENGTH = 10;
const int INITIAL_CYCLE_TEST_LENGTH = 3;

const int RESTART_DELAY = 800;
const int RESET_TL_LENGTH_EXTREMA_DELAY = 60;

const int MAX_ITERS = 12000;
const int SEMI_GREEDY_PARAM = 3;

int last_improvement_or_restart = 0;

int num_iters = 0;

bool non_tabu_moves = TRUE;
```

```
double best_makespan = HUGE_VAL;

bool use_N1 = FALSE;
bool use_N2 = FALSE;
bool use_NA = FALSE;
bool use_RNA = FALSE;
bool use_NB = FALSE;

int
main(int argc, const char** argv) {

  int numOperations;
  int numMachines;
  int numJobs;
  int numClasses;

  Job** jobs;
  Machine** machines;

  Operation** operations;

  double** classTransitions;

  if (argc > 2) {
    int idx = 1;
    while (idx < argc-1) {
      if (!strcasecmp(argv[idx], "-N1")) {
        use_N1 = TRUE;
      }
      else if (!strcasecmp(argv[idx], "-N2")) {
        use_N2 = TRUE;
      }
      else if (!strcasecmp(argv[idx], "-NA")) {
        use_NA = TRUE;
      }
      else if (!strcasecmp(argv[idx], "-RNA")) {
        use_RNA = TRUE;
      }
      else if (!strcasecmp(argv[idx], "-NB")) {
        use_NB = TRUE;
      }
      idx++;
    }
    cout << "using file " << argv[idx] << endl;
    parse(argv[idx], numJobs, jobs, numMachines, machines,
          numOperations, operations, numClasses, classTransitions);
  }
  else if (argc > 1) {
    cout << "using file " << argv[1] << endl;
    parse(argv[1], numJobs, jobs, numMachines, machines,
          numOperations, operations, numClasses, classTransitions);
    use_NB = TRUE;
    use_RNA = TRUE;
  }
  else {
    cout << "Usage: " << argv[0] << " [-N1] [-N2] [-NA] [-RNA] [-NB] <filename>" << endl;
    exit(0);
  }


  // we will need some random numbers
  srand48(time(NULL) ^ (getpid() + (getpid() << 15)));
  srand(time(NULL) ^ (getpid() + (getpid() << 15)));

  long start, end;
  time(&start);
  TS_Solution* ts = new TS_Solution(jobs, machines, classTransitions,
                                    numJobs, numMachines, numClasses);


  tabuSearchJS(ts);
  time(&end);

  cout << "execution took " << end-start << " seconds" << endl;
  cout << "best makespan: " << best_makespan << endl;
  cout << "------------------------------" << endl;

  delete ts;
};


/* ************************************************
 *
 * TABU SEARCH JS
 *
 * ************************************************/

void
tabuSearchJS(TS_Solution* ts) {                                        tabuSearchJS

  int i,j;

  int min, max; // lengths of Tabu List
  const OperationList* cp;

  int numMachines = ts->numMachines();

  OperationSig** solutionSig = new OperationSig*[numMachines];
  for (i = 0; i < numMachines; i++) {
    solutionSig[i] = new OperationSig[ts->mList(i)->numOperations()];
  }

  num_iters = 0;
  non_tabu_moves = TRUE;
  ts->tabuList()->updateLength(INITIAL_TABU_LENGTH);
  ts->witness()->setTimeToBreak(INITIAL_CYCLE_TEST_LENGTH);

  if (RESET_SOLUTIONS) {
```

```
  last_improvement_or_restart = 0;
  ts->tabuList()->reset();
  ts->witness()->reset();
}

initialSolution(ts);

double currMakespan, prevMakespan;
prevMakespan = best_makespan;

currMakespan = ts->makespan();
if (currMakespan < best_makespan) {
  best_makespan = currMakespan;

  for (i = 0; i < numMachines; i++) {
    for (j = 0; j < ts->mList(i)->numOperations(); j++) {
      solutionSig[i][j].job_ = ts->mList(i)->atRank(j)->job();
      solutionSig[i][j].jobIdx_ = ts->mList(i)->atRank(j)->jobIdx();
    }
  }
}

cp  = ts->computeCriticalPath(Operation::HEAD);

cout << "initial makespan: " << ts->makespan() << endl;

cp = ts->computeCriticalPath(Operation::TAIL);

while (keepSearching()) {

  if (ts->tabuList()->currentTime() % RESET_TL_LENGTH_EXTREMA_DELAY == 0) {
    min = (int)(drand48()*(ts->numMachines() + ts->numJobs()) / 3) + 2;
    max = (int)(drand48()*(ts->numMachines() + ts->numJobs()) / 3) + 6 + min;
  }

  if (RESET_SOLUTIONS &&
      (last_improvement_or_restart + RESTART_DELAY == ts->tabuList()->currentTime())) {

    for (i = 0; i < numMachines; i++) {
      for (j = 0; j < ts->mList(i)->numOperations(); j++) {
        ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_)->setMachine(i);
        ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_)->setMachineIdx(j);
        ts->mList(i)->setAtRank(j, ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_));
      }
    }

    ts->longestPathLinear(Operation::HEAD);
    ts->longestPathLinear(Operation::TAIL);
    cp = ts->computeCriticalPath(Operation::HEAD);

    last_improvement_or_restart = ts->tabuList()->currentTime();
    cout << "resetting the solution at time " << last_improvement_or_restart << endl;
  }
```

```
    exploreNeighborhood(ts);

    ts->longestPathLinear(Operation::HEAD);
    ts->longestPathLinear(Operation::TAIL);

    cp = ts->computeCriticalPath(Operation::HEAD);

    ts->tabuList()->incrementTime();
    num_iters++;

    currMakespan = ts->makespan();

    if (currMakespan < best_makespan) {
      best_makespan = currMakespan;

      for (i = 0; i < numMachines; i++) {
        for (j = 0; j < ts->mList(i)->numOperations(); j++) {
          solutionSig[i][j].job_ = ts->mList(i)->atRank(j)->job();
          solutionSig[i][j].jobIdx_ = ts->mList(i)->atRank(j)->jobIdx();
        }
      }
      ts->tabuList()->updateLength(1);
      cout << "found solution of length " << best_makespan
           << " at time " << ts->tabuList()->currentTime() << endl;

      if (RESET_SOLUTIONS) {
        last_improvement_or_restart = ts->tabuList()->currentTime();
      }
    }
    else {
      if (prevMakespan <= currMakespan && ts->tabuList()->length() < max) {
        ts->tabuList()->updateLength(ts->tabuList()->length() + 1);
      }
      else if (prevMakespan > currMakespan && ts->tabuList()->length() > min) {
        ts->tabuList()->updateLength(ts->tabuList()->length() - 1);
      }
    }

    prevMakespan = currMakespan;
  }

  cout << "##################################################" << endl;
  cout << best_makespan << endl;
  cout << "##################################################" << endl;


  for (i = 0; i < numMachines; i++) {
    for (j = 0; j < ts->mList(i)->numOperations(); j++) {
      ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_)->setMachine(i);
      ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_)->setMachineIdx(j);
      ts->mList(i)->setAtRank(j, ts->jList(solutionSig[i][j].job_)->atRank(solutionSig[i][j].jobIdx_));
    }
  }
  ts−>longestPathLinear(Operation::HEAD);
  ts−>longestPathLinear(Operation::TAIL);


  cp = ts−>computeCriticalPath(Operation::HEAD);

  for (i = 0; i < numMachines; i++) {
    delete [] solutionSig[i];
  }
  delete [] solutionSig;
};


/* ************************************************
 *
 * ESTIMATE LONGEST PATH
 *
 * ************************************************/

double
estimateLongestPath(Operation* sNode, Operation* eNode,                    estimateLonges
                    int& permutation, TS_Solution& tss) {

  double bestVal = HUGE_VAL;
  double currVal;
  int bestIdx = 0;

  Operation* start = sNode;
  Operation* end = eNode;

  Machine* m = tss.mList(start−>machine());
  int startIdx = start−>machineIdx();


  /*
    Permutations:
    1: ( end, start, )
    2: ( end, PM[start], start, )
    3: ( end, start, PM[start], )
    4: ( end, SM[end], start, )
    5: ( SM[end], end, start, )
   */


  tss.swap(m−>atRank(startIdx), m−>atRank(startIdx+1));

  bestIdx = 1;
  bestVal = longestPath(m, startIdx, 2, tss);

  // undo swap
  tss.swap(m−>atRank(startIdx), m−>atRank(startIdx+1));


  // arc must exist AND be on critical path!!
```

```
if (start->machineIdx() > 0 &&
    isOnCriticalPath(m->atRank(startIdx-1), tss.makespan()) &&
    (m->atRank(startIdx)->cumulativeTime(Operation::HEAD) ==
     m->atRank(startIdx-1)->cumulativeTime(Operation::HEAD) +
     m->atRank(startIdx-1)->time() + m->atRank(startIdx-1)->transitionTime())) {

  // set up test 2
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
  tss.swap(m->atRank(startIdx-1), m->atRank(startIdx));

  // test value
  currVal = longestPath(m, startIdx-1, 3, tss);

  // undo swaps
  tss.swap(m->atRank(startIdx-1), m->atRank(startIdx));
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

  if (currVal < bestVal) {
    bestIdx = 2;
    bestVal = currVal;
  }

  // set up test 3
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
  tss.swap(m->atRank(startIdx-1), m->atRank(startIdx));
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

  // test value
  currVal = longestPath(m, startIdx-1, 3, tss);

  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
  tss.swap(m->atRank(startIdx-1), m->atRank(startIdx));
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

  if (currVal< bestVal) {
    bestIdx = 3;
    bestVal = currVal;
  }
}

// arc must exist AND be on critical path!!
if (startIdx < tss.mList(start->machine())->numOperations() - 2 &&
    isOnCriticalPath(m->atRank(startIdx+2), tss.makespan()) &&
    (m->atRank(startIdx+2)->cumulativeTime(Operation::HEAD) ==
     m->atRank(startIdx+1)->cumulativeTime(Operation::HEAD) +
     m->atRank(startIdx+1)->time() + m->atRank(startIdx+1)->transitionTime()) ) {

  // set up test 4
  tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
  tss.swap(m->atRank(startIdx+1), m->atRank(startIdx+2));

  currVal = longestPath(m, startIdx, 3, tss);

  // undo swaps
```

```
    tss.swap(m->atRank(startIdx+1), m->atRank(startIdx+2));
    tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

    if (currVal < bestVal) {
      bestIdx = 4;
      bestVal = currVal;
    }

    // set up test 5
    tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
    tss.swap(m->atRank(startIdx+1), m->atRank(startIdx+2));
    tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

    currVal = longestPath(m, startIdx, 3, tss);

    // undo swaps
    tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));
    tss.swap(m->atRank(startIdx+1), m->atRank(startIdx+2));
    tss.swap(m->atRank(startIdx), m->atRank(startIdx+1));

    if (currVal < bestVal) {
      bestIdx = 5;
      bestVal = currVal;
    }
  }

  permutation = bestIdx;
  return bestVal;
};


/* ************************************************
 *
 * LONGEST PATH
 *
 * ************************************************/
double
longestPath(const Machine* const m, int startIdx, int num, const TS_Solution& tss) {          longestPath


  int i;

  int numNodes = num;

  double* newHeadVals = (double*)(alloca(numNodes*sizeof(double)));
  double* newTailVals = (double*)(alloca(numNodes*sizeof(double)));

  double jVal;
  double mVal;

  // comptute the new head values

  // special case for the first new head value
  if (startIdx > 0) {
```

```
  mVal = (m−>atRank(startIdx−1)−>cumulativeTime(Operation::HEAD) +
          m−>atRank(startIdx−1)−>time() +
          m−>atRank(startIdx−1)−>transitionTime()); // Previous Operation in Machine
}
else {
  mVal = 0;
}

if (tss.jobPrev(m−>atRank(startIdx)) != NULL) {
  jVal = (tss.jobPrev(m−>atRank(startIdx))−>cumulativeTime(Operation::HEAD) +
          tss.jobPrev(m−>atRank(startIdx))−>time()); // Previous Operation in Job
}
else {
  jVal = 0;
}

newHeadVals[0] = MAX(mVal, jVal);

for (i = 1; i < numNodes; i++) {
  if (tss.jobPrev(m−>atRank(startIdx+i)) != NULL) {
    jVal = (tss.jobPrev(m−>atRank(startIdx+i))−>cumulativeTime(Operation::HEAD) +
            tss.jobPrev(m−>atRank(startIdx+i))−>time()); // Previous Operation in Job

  }
  else {
    jVal = 0;
  }

  mVal = (newHeadVals[i−1] + m−>atRank(startIdx+i−1)−>time() +
          m−>atRank(startIdx+i−1)−>transitionTime()); // Previous Operation in Machine

  newHeadVals[i] = MAX(mVal, jVal);
}


// comptute the new tail values

// special case for the last new tail value
if (startIdx + numNodes < m−>numOperations()) {
  mVal = (m−>atRank(startIdx+numNodes)−>cumulativeTime(Operation::TAIL) +
          m−>atRank(startIdx+numNodes)−>time() +
          m−>atRank(startIdx+numNodes−1)−>transitionTime());
}
else {
  mVal = 0;
}

if (tss.jobNext(m−>atRank(startIdx+numNodes−1)) != NULL) {
  jVal = (tss.jobNext(m−>atRank(startIdx+numNodes−1))−>cumulativeTime(Operation::TAIL) +
          tss.jobNext(m−>atRank(startIdx+numNodes−1))−>time());
}
else {
  jVal = 0;
}
```

```
  newTailVals[numNodes−1] = MAX(mVal, jVal);

  for (i = numNodes − 2; i >= 0; i−−) {

    if (tss.jobNext(m−>atRank(startIdx + i)) != NULL) {
      jVal = (tss.jobNext(m−>atRank(startIdx + i))−>cumulativeTime(Operation::TAIL) +
              tss.jobNext(m−>atRank(startIdx + i))−>time());
    }
    else {
      jVal = 0;
    }

    mVal = (newTailVals[i+1] +
            m−>atRank(startIdx+i+1)−>time() +
            m−>atRank(startIdx+i)−>transitionTime());

    newTailVals[i] = MAX(jVal, mVal);
  }

  double toReturn = 0;

  for (i = 0; i < numNodes; i++) {
    toReturn = MAX(newHeadVals[i] + m−>atRank(startIdx+i)−>time() + newTailVals[i], toReturn);
  }

  return toReturn;
}


/* ************************************************
 *
 * INITIAL SOLUTION
 *
 * ************************************************/

void
initialSolution(TS_Solution* sol) {                                                    initialSolution


  int i,j;
  // Initialization:

  OperationList S, T;

  for (i = 0; i < sol−>numJobs(); i++) {
    S.addFirst(sol−>jList(i)−>atRank(0));
    S.first()−>data()−>setCumulativeTime(Operation::HEAD, 0);

    T.addFirst(sol−>jList(i)−>atRank(sol−>jList(i)−>numOperations() − 1));
    T.first()−>data()−>setCumulativeTime(Operation::TAIL, 0);
  }

  // free spots on machines
```

```
int* firstFree = new int[sol->numMachines()];
int* lastFree = new int[sol->numMachines()];

for (i = 0; i < sol->numMachines(); i++) {
  firstFree[i] = 0;
  lastFree[i] = (sol->mList(i))->numOperations() - 1;
}

// operations in jobs that have already been scheduled.
int* lastOperationInL = new int[sol->numJobs()];
int* firstOperationInR = new int[sol->numJobs()];

for (i = 0; i < sol->numJobs(); i++) {
  lastOperationInL[i] = -1;
  firstOperationInR[i] = sol->jList(i)->numOperations();
}

double* estimate = new double[sol->numJobs()];

int sizeOfL = 0;
int sizeOfR = 0;
int N = 0;
for (i = 0; i < sol->numJobs(); i++) {
  N += sol->jList(i)->numOperations();
}
// main algorithm
Operation* choice;
Operation* iData;
OperationList::Node* iter;

Machine* m;

double mVal, jVal; // used to compute head or tail values.

while (sizeOfR + sizeOfL < N) {

  int idx = 0;
  int mIdx;

  // choose some Operation \in S with a priority rule

  for (iter = S.first(); iter != NULL; iter = S.next(iter)) {
    iData = iter->data();

    sol->swap(sol->mList(iData->machine())->atRank(firstFree[iData->machine()]), iData);

    estimate[idx] = iData->time() + iData->cumulativeTime(Operation::HEAD);

    jVal = iData->cumulativeTime(Operation::TAIL);

    mVal = 0.0;
    // compute mVal here.
    for (mIdx = firstFree[iData->machine()]+1; mIdx <= lastFree[iData->machine()]; mIdx++) {
      m = sol->mList(iData->machine());
    mVal = MAX(mVal,
              m->atRank(mIdx)->time() +
              sol->transitionTime(iData->operationClass(), m->atRank(mIdx)->operationClass()) +
              sol->mList(iData->machine())->atRank(mIdx)->cumulativeTime(Operation::TAIL));
  }

  estimate[idx] += MAX(jVal, mVal);

  idx++;

  sol->swap(sol->mList(iData->machine())->atRank(firstFree[iData->machine()]), iData);
}

int choiceIdx = semiGreedy(SEMI_GREEDY_PARAM, estimate, S.size());

choice = S.atRank(choiceIdx)->data();

// put choice on machine in the first position free from the beginning

sol->swap(sol->mList(choice->machine())->atRank(firstFree[choice->machine()]), choice);
firstFree[choice->machine()]++;

// update sets
S.removeItem(choice);
sizeOfL++;
lastOperationInL[choice->job()]++;

// remove the Operation from T, if it is there.
if (T.findItem(choice)) {
  T.removeItem(choice);
}

if (firstOperationInR[choice->job()] > choice->jobIdx() + 1) {
  S.addFirst(sol->jobNext(choice));
}

// compute the head values of the elements in S

for (i = 0; i < sol->numJobs(); i++) {
  for (j = lastOperationInL[i] + 1; j < firstOperationInR[i]; j++) {
    sol->jList(i)->atRank(j)->setCumulativeTime(Operation::TAIL, -HUGE_VAL);
  }
}

for (iter = S.first(); iter != NULL; iter = S.next(iter)) {
  iter->data()->setCumulativeTime(Operation::HEAD, -HUGE_VAL);
}
for (iter = S.first(); iter != NULL; iter = S.next(iter)) {
  iData = iter->data();

  sol->swap(sol->mList(iData->machine())->atRank(firstFree[iData->machine()]), iData);
  sol->longestPathHelper(iData, Operation::HEAD);
  sol->longestPathHelperIncomplete(iData, Operation::TAIL, firstFree, lastFree);
```

```
    sol->swap(sol->mList(iData->machine())->atRank(firstFree[iData->machine()]), iData);
  }

  if (sizeOfL + sizeOfR < N) {

    // choose some Operation \in T with a priority rule
    idx = 0;
    for (iter = T.first(); iter != NULL; iter = T.next(iter)) {
      iData = iter->data();

      sol->swap(sol->mList(iData->machine())->atRank(lastFree[iData->machine()]), iData);
      estimate[idx] = iData->time() + iData->cumulativeTime(Operation::TAIL);

      jVal = iData->cumulativeTime(Operation::HEAD);

      mVal = 0.0;
      // compute mVal here.
      for (mIdx = firstFree[iData->machine()]; mIdx < lastFree[iData->machine()]; mIdx++) {
        mVal = MAX(mVal,
                   sol->mList(iData->machine())->atRank(mIdx)->time() +
                   sol->transitionTime(m->atRank(mIdx)->operationClass(), iData->operationClass()) +
                   sol->mList(iData->machine())->atRank(mIdx)->cumulativeTime(Operation::HEAD));
      }

      estimate[idx] += MAX(jVal, mVal);

      idx++;

      sol->swap(sol->mList(iData->machine())->atRank(lastFree[iData->machine()]), iData);
    }

    choiceIdx = semiGreedy(SEMI_GREEDY_PARAM, estimate, T.size());

    choice = T.atRank(choiceIdx)->data();

    // put i on machine_i in the first position free from the end
    sol->swap(sol->mList(choice->machine())->atRank(lastFree[choice->machine()]), choice);
    lastFree[choice->machine()]--;

    // update the sets
    T.removeItem(choice);
    sizeOfR++;
    firstOperationInR[choice->job()]--;

    if (S.findItem(choice)) {
      S.removeItem(choice);
    }

    if (lastOperationInL[choice->job()] < choice->jobIdx() - 1) {
      T.addFirst(sol->jobPrev(choice));
    }
  }
  // compute the tail values of the elements in T
```

```
    for (i = 0; i < sol->numJobs(); i++) {
      for (j = lastOperationInL[i] + 1; j < firstOperationInR[i]; j++) {
        sol->jList(i)->atRank(j)->setCumulativeTime(Operation::HEAD, −HUGE_VAL);
      }
    }

    for (iter = T.first(); iter != NULL; iter = T.next(iter)) {
      iter->data()->setCumulativeTime(Operation::TAIL, −HUGE_VAL);
    }
    for (iter = T.first(); iter != NULL; iter = T.next(iter)) {
      iData = iter->data();

      sol->swap(sol->mList(iData->machine())->atRank(lastFree[iData->machine()]), iData);
      sol->longestPathHelper(iData, Operation::TAIL);
      sol->longestPathHelperIncomplete(iData, Operation::HEAD, firstFree, lastFree);
      sol->swap(sol->mList(iData->machine())->atRank(lastFree[iData->machine()]), iData);
    }

  }

  sol->longestPathLinear(Operation::TAIL);
  sol->longestPathLinear(Operation::HEAD);

  delete [] firstOperationInR;
  delete [] lastOperationInL;

  delete [] firstFree;
  delete [] lastFree;

  delete [] estimate;

};


/* ************************************************
 *
 * EXPLORE NEIGHBORHOOD
 *
 * ************************************************/

void
exploreNeighborhood(TS_Solution* sol) {                                    exploreNeighbo
  // this function should explore the given neighborhood and return
  // the appropriate candidate solution.

  // NT means Non Tabu
  NeighboringSolutions nt;

  nt.hasMove_  = FALSE;
  nt.bestMove_ = HUGE_VAL;
  nt.moveIsNA_ = FALSE;

  nt.naPermutation_ = −1;
  nt.start_         = NULL;
nt.end_             = NULL;

nt.toMove_      = −1;
nt.destination_ = −1;
nt.toModify_    = −1;

// rand means random move. only used if no move is non-tabu or meets
// the aspiration criterion.
NeighboringSolutions rand;

rand.hasMove_  = FALSE;
rand.bestMove_ = HUGE_VAL;
rand.moveIsNA_ = FALSE;

rand.naPermutation_ = −1;
rand.start_         = NULL;
rand.end_           = NULL;

rand.toMove_      = −1;
rand.destination_ = −1;
rand.toModify_    = −1;


// num is used to break ties when two solutions have the same
// estimated value
double num;

// we will want to have a reserve move in case there exist no
// non-tabu moves, and none of the non-tabu moves satisfy the
// aspiration criterion.
double reserveNum = 1.0;

//Here is where the magic happens.
if (use_N1) {
  n1(nt, rand, sol, num, reserveNum);
}
if (use_N2) {
  n1(nt, rand, sol, num, reserveNum);
}
if (use_NA) {
  na(nt, rand, sol, num, reserveNum);
}
if (use_RNA) {
  rna(nt, rand, sol, num, reserveNum);
}
if (use_NB) {
  nb(nt, rand, sol, num, reserveNum);
}

NeighboringSolutions toUse;
if (nt.hasMove_) {
  toUse = nt;
}
else if (rand.hasMove_) {
    toUse = rand;
  }
  else {
    // there are no valid moves
    non_tabu_moves = FALSE;
    return;
  }

  if (toUse.moveIsNA_) {
    /*
      Permutations indicated from estimateLongestPath(.) :

      1: (PM[start], end, start, SM[end])
      2: (PM[PM[start]], end, PM[start], start, SM[end])
      3: (PM[PM[start]], end, start, PM[start], SM[end])
      4: (PM[start], end, SM[end], start, SM[SM[end])
      5: (PM[start], SM[end], end, start, SM[SM[end])
    */
    Operation* startOp = toUse.start_;
    Operation* endOp = toUse.end_;

    Operation* thirdOp; // used if swapping three elements
    Machine* m = sol->mList(startOp->machine());

    sol->witness()->adjustCycleDepth(sol->witness()->query(startOp, endOp, sol->makespan()));
    sol->witness()->mark(startOp, endOp, sol->makespan());//witness arc
    //
    // Now to update the machine list and tabu list.
    //

    switch (toUse.naPermutation_) {
    case 1:
      //update tabulist
      sol->tabuList()->mark(endOp, startOp);

      // swap (start, end)
      sol->swap(startOp, endOp);
      break;
    case 2:
      thirdOp = sol->machinePrev(startOp);

      //update tabulist
      sol->tabuList()->mark(endOp, startOp);
      sol->tabuList()->mark(endOp, thirdOp);
      // the paper indicates that this should be here...
      sol->tabuList()->mark(startOp, thirdOp);

      // swap (start, end), (PM[start], end)
      sol->swap(startOp, endOp);
      sol->swap(endOp, thirdOp);
      break;
    case 3:
      thirdOp = sol->machinePrev(startOp);
      //update tabulist
      sol->tabuList()->mark(endOp, startOp);
      sol->tabuList()->mark(endOp, thirdOp);
      sol->tabuList()->mark(startOp, thirdOp);

      // swap (start, end), (PM[start], end), (PM[start], start)
      sol->swap(startOp, endOp);
      sol->swap(endOp, thirdOp);
      sol->swap(thirdOp, startOp);
      break;
    case 4:
      thirdOp = sol->machineNext(endOp);

      //update tabulist
      sol->tabuList()->mark(endOp, startOp);
      sol->tabuList()->mark(thirdOp, startOp);
      // the paper indicates that this should be here...
      sol->tabuList()->mark(thirdOp, endOp);

      // swap (start, end), (start, SM[end])
      sol->swap(startOp, endOp);
      sol->swap(startOp, thirdOp);
      break;
    case 5:
      thirdOp = sol->machineNext(endOp);

      //update tabulist
      sol->tabuList()->mark(endOp, startOp);
      sol->tabuList()->mark(thirdOp, startOp);
      sol->tabuList()->mark(thirdOp, endOp);

      // swap (start, end), (start, SM[end]), (end, SM[end])
      sol->swap(startOp, endOp);
      sol->swap(startOp, thirdOp);
      sol->swap(thirdOp, endOp);
      break;
    }
  }
  else { // using NB
    int destination    = toUse.destination_;
    int toMove         = toUse.toMove_;
    Machine* toModify = sol->mList(toUse.toModify_);
    CycleWitness* cw = sol->witness();

    Operation* temp = toModify->atRank(toMove);

    if (destination < toMove) {

      cw->adjustCycleDepth(cw->query(temp, toModify->atRank(toMove-1), sol->makespan()));
      cw->mark(temp, toModify->atRank(toMove-1), sol->makespan());//witness arc

      for (int i = toMove; i > destination; i--) {
        sol->tabuList()->mark(temp, toModify->atRank(i-1));
        toModify->setAtRank(i, toModify->atRank(i-1));
        toModify->atRank(i)->setMachineIdx(i);
      }
      toModify->setAtRank(destination, temp);
      toModify->atRank(destination)->setMachineIdx(destination);
    }
    else {

      cw->adjustCycleDepth(cw->query(toModify->atRank(toMove+1), temp, sol->makespan()));
      cw->mark(toModify->atRank(toMove+1), temp, sol->makespan());//witness arc

      for (int i = toMove; i < destination; i++) {
        sol->tabuList()->mark(toModify->atRank(i+1), temp);
        toModify->setAtRank(i, toModify->atRank(i+1));
        toModify->atRank(i)->setMachineIdx(i);
      }
      toModify->setAtRank(destination, temp);
      toModify->atRank(destination)->setMachineIdx(destination);
    }

  }

};



/* ************************************************
 *
 * TEST NB MOVE
 *
 * ************************************************/
double
testNBMove(Machine* m, int toMove, int destination, TS_Solution* sol) {          testNBMove
  int k;
  double toReturn;
  Operation* temp;

  if (destination > toMove) {
    temp = m->atRank(toMove);
    for (k = toMove; k < destination; k++) {
      m->setAtRank(k, m->atRank(k+1));
    }
    m->setAtRank(destination, temp);
    toReturn = longestPath(m, toMove, destination-toMove+1, *sol);

    temp = m->atRank(destination);
    for (k = destination; k > toMove; k--) {
      m->setAtRank(k, m->atRank(k-1));
    }
    m->setAtRank(toMove, temp);
  }
  else {
    temp = m->atRank(toMove);
    for (k = toMove; k > destination; k--) {
      m->setAtRank(k, m->atRank(k-1));
    }
    m->setAtRank(destination, temp);
    toReturn = longestPath(m, destination, toMove-destination+1, *sol);

    temp = m->atRank(destination);
    for (k = destination; k < toMove; k++) {
      m->setAtRank(k, m->atRank(k+1));
    }
    m->setAtRank(toMove, temp);

  }
  return toReturn;
}



/* ************************************************
 *
 * SEMI GREEDY
 *
 * ************************************************/

int
semiGreedy(int c, const double* const vals, int numVals) {                semiGreedy

  int i, j, k;

  if (numVals <= c) {
    return rand()%numVals;
  }
  else {
    int* cLowestOrderStatistics = (int*)(alloca(c*sizeof(int)));
    double* cLowestValues = (double*)(alloca(c*sizeof(double)));

    for (i = 0; i < c; i++) {
      cLowestValues[i] = HUGE_VAL;
    }

    for (i = 0; i < numVals; i++) {
      for (j = 0; j < c; j++) {
        if (vals[i] < cLowestValues[j]) {
          break;
        }
      }
      for (k = c-1; k >= j+1; k--) {
        cLowestValues[k] = cLowestValues[k-1];
        cLowestOrderStatistics[k] = cLowestOrderStatistics[k-1];
      }
      if (j < c) {
        cLowestValues[j] = vals[i];
        cLowestOrderStatistics[j] = i;
      }
    }
    int returnRank = rand()%c;
    return cLowestOrderStatistics[returnRank];
  }
}

/* ************************************************
 *
 * KEEP SEARCHING
 *
 * ************************************************/

bool
keepSearching() {
  return ((num_iters < MAX(MAX_ITERS, last_improvement_or_restart + RESTART_DELAY)) &&
          non_tabu_moves == TRUE );
}

/* ************************************************
 *
 * MEETS ASPIRATION CRITERION
 *
 * ************************************************/
bool
meetsAspirationCriterion(double estimate) {
  // If selected move improves better than best so far, accept.
  return (estimate < best_makespan);
};


void
print(const OperationList* const ol) {
  OperationList::Node* iter = ol->first();
  cout << "List: " << endl;
  while (iter != NULL) {

    iter->data()->dump();
    cout << endl;

    iter = ol->next(iter);
  }

}

/* ************************************************
 *
 * IS ON CRITICAL PATH
 *
 * ************************************************/
bool
isOnCriticalPath(const Operation* const toTest, double makespan) {
  // r_p + d_p + t_p == makespan ==> p \in critical path
  return (toTest->cumulativeTime(Operation::HEAD) + toTest->time() +
          toTest->cumulativeTime(Operation::TAIL) == makespan);
}
```

```
/* ************************************************
 *
 * NA MOVE IS NOT TABU
 *
 * ************************************************/
bool
naMoveIsNotTabu(Operation* start, int permutation, TS_Solution* sol) {
  /*
    Permutations indicated from estimateLongestPath(.) :

    1: (PM[start], end, start, SM[end])
    2: (PM[PM[start]], end, PM[start], start, SM[end])
    3: (PM[PM[start]], end, start, PM[start], SM[end])
    4: (PM[start], end, SM[end], start, SM[SM[end])
    5: (PM[start], SM[end], end, start, SM[SM[end])
   */

  bool isNotTabu = TRUE;

  int startIdx = start->machineIdx();
  Machine* m = sol->mList(start->machine());

  // query returns TRUE if a move is tabu.
  // all RNA tests reverse this arc. If it is tabu, no move is feasible.
  isNotTabu = !sol->tabuList()->query(m->atRank(startIdx), m->atRank(startIdx + 1));
  if (isNotTabu) {
    switch(permutation) {
    case 2:
      isNotTabu = isNotTabu &&
        !sol->tabuList()->query(m->atRank(startIdx-1), m->atRank(startIdx+1));
      break;
    case 3:
      isNotTabu = isNotTabu &&
        !sol->tabuList()->query(m->atRank(startIdx-1), m->atRank(startIdx+1)) &&
        !sol->tabuList()->query(m->atRank(startIdx-1), m->atRank(startIdx));
      break;
    case 4:
      isNotTabu = isNotTabu &&
        !sol->tabuList()->query(m->atRank(startIdx), m->atRank(startIdx+2));
      break;
    case 5:
      isNotTabu = isNotTabu &&
        !sol->tabuList()->query(m->atRank(startIdx), m->atRank(startIdx+2)) &&
        !sol->tabuList()->query(m->atRank(startIdx+1), m->atRank(startIdx+2));
      break;
    };
  }
  return isNotTabu;
}

/* ************************************************
 *
 * FILL NA SOLUTIONS
 *
 * ************************************************/
void
fillNASolutions(NeighboringSolutions& nt, NeighboringSolutions& rand,
                double currTest, int permutation, Operation* start, Operation* end,
                bool isCycle, bool isNotTabu, double& num, double& reserveNum) {

  if ((isNotTabu && !isCycle) || meetsAspirationCriterion(currTest)) {

    if (currTest < nt.bestMove_) {
      nt.hasMove_        = TRUE;
      nt.moveIsNA_       = TRUE;
      nt.start_          = start;
      nt.end_            = end;
      nt.naPermutation_ = permutation;
      nt.bestMove_       = currTest;

      num = 2.0;
    }
    else if (currTest == nt.bestMove_ && drand48() < 1/num) {
      nt.moveIsNA_       = TRUE;
      nt.start_          = start;
      nt.end_            = end;
      nt.naPermutation_ = permutation;

      num++;
    }
  }
  else if (!(nt.hasMove_) && drand48() < 1/reserveNum) {
    rand.hasMove_        = TRUE;
    rand.moveIsNA_       = TRUE;
    rand.start_          = start;
    rand.end_            = end;
    rand.naPermutation_ = permutation;

    reserveNum++;
  }
}

/* ************************************************
 *
 * FILL NB SOLUTIONS
 *
 * ************************************************/
void
fillNBSolutions(NeighboringSolutions& nt, NeighboringSolutions& rand,
               double currTest, int toMove, int destination, int toModify,
               bool isCycle, bool isNotTabu, double& num, double& reserveNum) {

  if ((isNotTabu && !isCycle) || meetsAspirationCriterion(currTest)) {

    if (currTest < nt.bestMove_) {
      nt.hasMove_        = TRUE;
      nt.moveIsNA_       = FALSE;
      nt.destination_    = destination;
      nt.toMove_         = toMove;
      nt.toModify_       = toModify;
      nt.bestMove_       = currTest;

      num = 2.0;
    }
    else if (currTest == nt.bestMove_ && drand48() < 1/num) {
      nt.moveIsNA_    = FALSE;
      nt.destination_    = destination;
      nt.toMove_         = toMove;
      nt.toModify_       = toModify;

      num++;
    }
  }
  else if (!(nt.hasMove_) && drand48() < 1/reserveNum) {
    rand.hasMove_         = TRUE;
    rand.moveIsNA_      = FALSE;
    rand.destination_      = destination;
    rand.toMove_           = toMove;
    rand.toModify_         = toModify;

    reserveNum++;
  }
}

/* ************************************************
 *
 * NEIGHBORHOOD NB
 *
 * *************************************************/
  /* ********************
   * Neighborhood NB
   * 1) Identify blocks.
   *
   * 2) For each block b found
   *    3) for each operation x in b
   *       4) for each k from PJ[x] to b.start
   *          5) if (head[SJ[k]] + time[SJ[k]] < head[PJ[x]])
   *             6) break;
   *          7) test move (x, SM[k])
   *       8) for each k from SJ[x] to b.end
   *          9) if (head[SJ[x]] + time[SJ[x]] < head[PJ[k]])
   *             10) break;
   *          11) test move (x, PM[k])
   * ********************/
void
nb(NeighboringSolutions& nt, NeighboringSolutions& rand,                    nb
     TS_Solution* sol, double& num, double& reserveNum) {


  static int i, j, k;
  static int fIdx, bIdx, blocksize;
  static double currTest;
  static bool isCycle, isNotTabu;
  static Machine* m;

  int offset = 1;
  if (use_RNA || use_NA) {
    offset = 3;
  }
  const int RNA_OFFSET = offset;

  for (int n = 0; n < sol->numMachines(); n++) {
    m = sol->mList(n);
    for (int p = 0; p < sol->mList(n)->numOperations(); p++) {
      bIdx = p;
      fIdx = p;

      // this is very important. it ensures that a block is actually
      // on the critical path.
      if (!isOnCriticalPath(m->atRank(p), sol->makespan())) {
        continue;
      }

      while (p < sol->mList(n)->numOperations()-1 &&
             isOnCriticalPath(m->atRank(p+1), sol->makespan()) &&
             m->atRank(p+1)->cumulativeTime(Operation::HEAD) ==
             m->atRank(p)->cumulativeTime(Operation::HEAD) +
             m->atRank(p)->time() + m->atRank(p)->transitionTime()) {
        p++;
      }

      fIdx = p;
      blocksize = fIdx - bIdx + 1;

      if (blocksize > RNA_OFFSET) { // else, everything is covered by RNA

        // try to swap operation to as low an index as possible
        for (i = bIdx + RNA_OFFSET; i <= fIdx; i++) {
          for (j = i-1; j > bIdx; j--) {

            if (sol->jobNext(m->atRank(j)) != NULL && sol->jobPrev(m->atRank(i)) != NULL &&
                sol->jobNext(m->atRank(j))->cumulativeTime(Operation::HEAD) +
                sol->jobNext(m->atRank(j))->time() <=
                sol->jobPrev(m->atRank(i))->cumulativeTime(Operation::HEAD)) {
              j++;

              break;
            }
          }


          if (j == bIdx &&
              sol->jobNext(m->atRank(j)) != NULL && sol->jobPrev(m->atRank(i)) != NULL &&
              sol->jobNext(m->atRank(j))->cumulativeTime(Operation::HEAD) +
              sol−>jobNext(m−>atRank(j))−>time() <=
              sol−>jobPrev(m−>atRank(i))−>cumulativeTime(Operation::HEAD)) {
            j++;
          }
          if (i == j) { continue; }

          isNotTabu = TRUE;

          for (k = i; k > j; k−−) {
            isNotTabu = isNotTabu && !(sol−>tabuList()−>query(m−>atRank(k−1), m−>atRank(i)));
          }

          isCycle = sol−>witness()−>isInCycle() &&
            sol−>witness()−>query(m−>atRank(i), m−>atRank(i−1), sol−>makespan());

          currTest = testNBMove(m, i, j, sol);

          fillNBSolutions(nt, rand, currTest, i, j, n, isCycle, isNotTabu, num, reserveNum);

        }
        // end swap to low

        // try to swap operations to as high an index as possible
        for (i = bIdx; i <= fIdx − RNA_OFFSET; i++) {
          for (j = i+1; j < fIdx; j++) {

            if (sol−>jobNext(m−>atRank(i)) != NULL && sol−>jobPrev(m−>atRank(j)) != NULL &&
                sol−>jobNext(m−>atRank(i))−>cumulativeTime(Operation::HEAD) +
                sol−>jobNext(m−>atRank(i))−>time() <=
                sol−>jobPrev(m−>atRank(j))−>cumulativeTime(Operation::HEAD)) {
              j−−;

              break;
            }
          }


          if (j == fIdx &&
              sol−>jobNext(m−>atRank(i)) != NULL && sol−>jobPrev(m−>atRank(j)) != NULL &&
              sol−>jobNext(m−>atRank(i))−>cumulativeTime(Operation::HEAD) +
              sol−>jobNext(m−>atRank(i))−>time() <=
              sol−>jobPrev(m−>atRank(j))−>cumulativeTime(Operation::HEAD)) {
            j−−;
          }
          if (i == j) { continue; }

          //test move
          isNotTabu = TRUE;

          for (k = j; k > i; k−−) {
            isNotTabu = isNotTabu && !sol−>tabuList()−>query(m−>atRank(i), m−>atRank(k));
          }

          isCycle = sol−>witness()−>isInCycle() &&
              sol->witness()->query(m->atRank(i), m->atRank(i+1), sol->makespan());

            currTest = testNBMove(m, i, j, sol);

            fillNBSolutions(nt, rand, currTest, i, j, n, isCycle, isNotTabu, num, reserveNum);

          }
          // end swap to high
        }
      }
    }
  }


/* ************************************************
 *
 * NEIGHBORHOOD RNA
 *
 * ************************************************/

  /* ********************************************************
   * RNA:
   *
   * for each pair of consecutive cp operations belonging to the same machine {
   *   if (!(PM[start] \in cp \AND SM[end] \in cp)) {
   *     if (estimateLongestPath(.) > bestSoFar) {
   *       bestSoFar <- estimateLongestPath(.);
   *       store Nodes;
   *       store # of permutation;
   *     }
   *   }
   * }
   * ********************************************************/
void
rna(NeighboringSolutions& nt, NeighboringSolutions& rand,                    rna
    TS_Solution* sol, double& num, double& reserveNum) {

  int permutation, fIdx, bIdx, blocksize;
  double currTest;
  bool isCycle, isNotTabu;
  Machine* m;

  for (int n = 0; n < sol->numMachines(); n++) {
    m = sol->mList(n);

    for (int p = 0; p < sol->mList(n)->numOperations(); p++) {
      bIdx = p;
      fIdx = p;

      // this is very important. it ensures that a block is actually
      // on the critical path.
      if (!isOnCriticalPath(m->atRank(p), sol->makespan())) {
        continue;
      }
      while (p < sol->mList(n)->numOperations()-1 &&
             isOnCriticalPath(m->atRank(p+1), sol->makespan()) &&
             m->atRank(p+1)->cumulativeTime(Operation::HEAD) ==
             m->atRank(p)->cumulativeTime(Operation::HEAD) +
             m->atRank(p)->time() + m->atRank(p)->transitionTime()) {
        p++;
      }

      fIdx = p;
      blocksize = fIdx - bIdx + 1;
      if (blocksize > 1) {
        currTest = estimateLongestPath(m->atRank(bIdx), m->atRank(bIdx+1), permutation, *sol);
        // test the first arc
        isCycle = sol->witness()->isInCycle() &&
          sol->witness()->query(m->atRank(bIdx), m->atRank(bIdx+1), sol->makespan());

        //we only test arcs in the RNA neighborhood, so we don't need
        //to check this condition.

        isNotTabu = naMoveIsNotTabu(m->atRank(bIdx), permutation, sol);

        fillNASolutions(nt, rand, currTest, permutation,
                        m->atRank(bIdx), m->atRank(bIdx+1),
                        isCycle, isNotTabu, num, reserveNum);

        if (blocksize > 2) {

          currTest = estimateLongestPath(m->atRank(fIdx-1), m->atRank(fIdx), permutation, *sol);

          // test the second arc
          isCycle = sol->witness()->isInCycle() &&
            sol->witness()->query(m->atRank(fIdx-1), m->atRank(fIdx), sol->makespan());

          //we only test arcs in the RNA neighborhood, so we don't
          //need to check this condition.

          isNotTabu = naMoveIsNotTabu(m->atRank(fIdx-1), permutation, sol);

          fillNASolutions(nt, rand, currTest, permutation,
                          m->atRank(fIdx-1), m->atRank(fIdx),
                          isCycle, isNotTabu, num, reserveNum);
        }
      }
    }
  }
}


/* **************************************************
 *
 * NEIGHBORHOOD NA
 *
 * *************************************************/
```

```
  /* ************************************************************
   * NA:
   *
   * for each pair of consecutive cp operations belonging to the same machine {
   *   if (estimateLongestPath(.) < bestSoFar) {
   *       bestSoFar <- estimateLongestPath(.);
   *       store Nodes;
   *       store # of permutation;
   *    }
   * }
   * ************************************************************/
void
na(NeighboringSolutions& nt, NeighboringSolutions& rand,                                  na
   TS_Solution* sol, double& num, double& reserveNum) {

  int permutation, fIdx, bIdx, blocksize;
  double currTest;
  bool isCycle, isNotTabu;
  Machine* m;

  for (int n = 0; n < sol->numMachines(); n++) {
    m = sol->mList(n);

    for (int p = 0; p < sol->mList(n)->numOperations(); p++) {
      bIdx = p;
      fIdx = p;

      // this is very important. it ensures that a block is actually
      // on the critical path.
      if (!isOnCriticalPath(m->atRank(p), sol->makespan())) {
        continue;
      }

      while (p < sol->mList(n)->numOperations()-1 &&
             isOnCriticalPath(m->atRank(p+1), sol->makespan()) &&
             m->atRank(p+1)->cumulativeTime(Operation::HEAD) ==
             m->atRank(p)->cumulativeTime(Operation::HEAD) +
             m->atRank(p)->time() + m->atRank(p)->transitionTime()) {
        p++;
      }

      fIdx = p;
      blocksize = fIdx - bIdx + 1;
      if (blocksize > 1) {
        for (int start = bIdx; start < fIdx; start++) {
          currTest = estimateLongestPath(m->atRank(start), m->atRank(start+1), permutation, *sol);

          // test the first arc
          isCycle = sol->witness()->isInCycle() &&
            sol->witness()->query(m->atRank(start), m->atRank(start+1), sol->makespan());

          isNotTabu = naMoveIsNotTabu(m->atRank(start), permutation, sol);
          fillNASolutions(nt, rand, currTest, permutation,
                         m->atRank(start), m->atRank(start+1),
                         isCycle, isNotTabu, num, reserveNum);
        }
      }
    }
  }
}

/* ************************************************
 *
 * NEIGHBORHOOD N1
 *
 * ************************************************/

  /* ********************************************************
   *  N1:
   *
   *  for each pair of consecutive cp operations belonging to the same machine {
   *    if (estimateLongestPath(.) < bestSoFar) {
   *        bestSoFar <- estimateLongestPath(.);
   *        store Nodes;
   *        store # of permutation;
   *     }
   *  }
   * ********************************************************/
void
n1(NeighboringSolutions& nt, NeighboringSolutions& rand,                    n1
   TS_Solution* sol, double& num, double& reserveNum) {

  int fIdx, bIdx, blocksize;
  double currTest;
  bool isCycle, isNotTabu;
  Machine* m;

  for (int n = 0; n < sol->numMachines(); n++) {
    m = sol->mList(n);

    for (int p = 0; p < sol->mList(n)->numOperations(); p++) {
      bIdx = p;
      fIdx = p;

      // this is very important. it ensures that a block is actually
      // on the critical path.
      if (!isOnCriticalPath(m->atRank(p), sol->makespan())) {
        continue;
      }

      while (p < sol->mList(n)->numOperations()-1 &&
             isOnCriticalPath(m->atRank(p+1), sol->makespan()) &&
             m->atRank(p+1)->cumulativeTime(Operation::HEAD) ==
             m->atRank(p)->cumulativeTime(Operation::HEAD) +
             m->atRank(p)->time() + m->atRank(p)->transitionTime()) {
        p++;
      }

      fIdx = p;
      blocksize = fIdx − bIdx + 1;
      if (blocksize > 1) {
        for (int start = bIdx; start < fIdx; start++) {
          sol−>swap(m−>atRank(start), m−>atRank(start+1));
          currTest = longestPath(m, start, 2, *sol);
          // undo swap
          sol−>swap(m−>atRank(start), m−>atRank(start+1));

          // test the first arc
          isCycle = sol−>witness()−>isInCycle() &&
            sol−>witness()−>query(m−>atRank(start), m−>atRank(start+1), sol−>makespan());

          isNotTabu = naMoveIsNotTabu(m−>atRank(start), 1, sol);

          fillNASolutions(nt, rand, currTest, 1,
                          m−>atRank(start), m−>atRank(start+1),
                          isCycle, isNotTabu, num, reserveNum);
        }
      }
    }
  }
}


/* ************************************************
 *
 * NEIGHBORHOOD N2
 *
 * ************************************************/
void
n2(NeighboringSolutions& nt, NeighboringSolutions& rand,                     n2
   TS_Solution* sol, double& num, double& reserveNum) {

  int fIdx, bIdx, blocksize;
  double currTest;
  bool isCycle, isNotTabu;
  Machine* m;

  for (int n = 0; n < sol−>numMachines(); n++) {
    m = sol−>mList(n);

    for (int p = 0; p < sol−>mList(n)−>numOperations(); p++) {
      bIdx = p;
      fIdx = p;

      // this is very important. it ensures that a block is actually
      // on the critical path.
      if (!isOnCriticalPath(m−>atRank(p), sol−>makespan())) {
        continue;
      }
      while (p < sol->mList(n)->numOperations()-1 &&
             isOnCriticalPath(m->atRank(p+1), sol->makespan()) &&
             m->atRank(p+1)->cumulativeTime(Operation::HEAD) ==
             m->atRank(p)->cumulativeTime(Operation::HEAD) +
             m->atRank(p)->time() + m->atRank(p)->transitionTime()) {
        p++;
      }

      fIdx = p;
      blocksize = fIdx - bIdx + 1;
      if (blocksize > 1) {
        sol->swap(m->atRank(bIdx), m->atRank(bIdx+1));
        currTest = longestPath(m, bIdx, 2, *sol);
        // undo swap
        sol->swap(m->atRank(bIdx), m->atRank(bIdx+1));

        // test the first arc
        isCycle = sol->witness()->isInCycle() &&
          sol->witness()->query(m->atRank(bIdx), m->atRank(bIdx+1), sol->makespan());

        isNotTabu = naMoveIsNotTabu(m->atRank(bIdx), 1, sol);

        fillNASolutions(nt, rand, currTest, 1,
                        m->atRank(bIdx), m->atRank(bIdx+1),
                        isCycle, isNotTabu, num, reserveNum);

        if (blocksize > 2) {
          sol->swap(m->atRank(fIdx-1), m->atRank(fIdx));
          currTest = longestPath(m, fIdx-1, 2, *sol);
          // undo swap
          sol->swap(m->atRank(fIdx-1), m->atRank(fIdx));

          // test the first arc
          isCycle = sol->witness()->isInCycle() &&
            sol->witness()->query(m->atRank(fIdx-1), m->atRank(fIdx), sol->makespan());

          isNotTabu = naMoveIsNotTabu(m->atRank(fIdx-1), 1, sol);

          fillNASolutions(nt, rand, currTest, 1,
                          m->atRank(fIdx-1), m->atRank(fIdx),
                          isCycle, isNotTabu, num, reserveNum);

        }
      }
    }
  }
}


/* ***********************************************
 *
 * INITIAL SOLUTION 2
 *
 * ***********************************************/
```

```
void
initialSolution2(TS_Solution* sol) {                                    initialSolution2

  int i,j;
  // Initialization:

  OperationList S;

  for (i = 0; i < sol->numJobs(); i++) {
    S.addFirst(sol->jList(i)->atRank(0));
    S.first()->data()->setCumulativeTime(Operation::HEAD, 0);
  }

  int* firstFree = new int[sol->numMachines()];
  int* lastFree = new int[sol->numMachines()];

  for (i = 0; i < sol->numMachines(); i++) {
    firstFree[i] = 0;
    lastFree[i] = (sol->mList(i))->numOperations() - 1;
  }

  int* lastOperationInL = new int[sol->numJobs()];
  int* firstOperationInR = new int[sol->numJobs()];

  for (i = 0; i < sol->numJobs(); i++) {
    lastOperationInL[i] = -1;
    firstOperationInR[i] = sol->jList(i)->numOperations();
  }

  int sizeOfL = 0;
  int N = 0;
  for (i = 0; i < sol->numJobs(); i++) {
    N += sol->jList(i)->numOperations();
  }

  // main algorithm
  while (sizeOfL < N) {
    Operation* choice;
    int swapIdx;
    double bestTime = -HUGE_VAL;
    OperationList::Node* iter;
    double num;

    // compute the work remaining for each x \in S
    for (i = 0; i < sol->numJobs(); i++) {
      for (j = lastOperationInL[i] + 1; j < firstOperationInR[i]; j++) {
        sol->jList(i)->atRank(j)->setCumulativeTime(Operation::TAIL, -HUGE_VAL);
      }
    }
    for (iter = S.first(); iter != NULL; iter = S.next(iter)) {
      sol->longestPathHelperIncomplete(iter->data(), Operation::TAIL, firstFree, lastFree);
    }
```

```
      // choose some Operation \in S with MWKR priority rule
      for (iter = S.first(); iter != NULL; iter = S.next(iter)) {
        if (iter->data()->cumulativeTime(Operation::TAIL) > bestTime) {
          bestTime = iter->data()->cumulativeTime(Operation::TAIL);
          choice = iter->data();
          num = 2.0;
        }
        else if (iter->data()->cumulativeTime(Operation::TAIL) == bestTime) {
          if (drand48() < 1/num) {
            choice = iter->data();
            num++;
          }
        }

      }

      // put choice on machine in the first position free from the beginning
      choice->setMachineIdx(firstFree[choice->machine()]);
      sol->mList(choice->machine())->setAtRank(firstFree[choice->machine()], choice);
      firstFree[choice->machine()]++;

      // update sets
      S.removeItem(choice);
      sizeOfL++;
      lastOperationInL[choice->job()]++;

      if (firstOperationInR[choice->job()] > choice->jobIdx() + 1) {
        S.addFirst(sol->jobNext(choice));
      }
    }

  sol->longestPathLinear(Operation::TAIL);
  sol->longestPathLinear(Operation::HEAD);

  delete [] firstOperationInR;
  delete [] lastOperationInL;

  delete [] firstFree;
  delete [] lastFree;
};
```

B Test Instances

In all of the following test instances, the number of operations, operation classes, jobs and machines are given by their respective tags. Following the initial tags, there is a line of text for each job. On each job, the operations are defined as a triple of numbers. The first number is the index (0 to NUM_MACHINES-1) of the machine on which the operation is to be executed. The second number is the duration of the operation, and the third number is the index of the class to which it belongs. Following the operation data is a NUM_CLASSES x NUM_CLASSES matrix of the setup times where the element ij denotes the time to setup when changing from an operation in class i to an operation in class j.

B.1 MT6-TT

```
NUM_OPERATIONS 36
NUM_CLASSES 3
NUM_JOBS 6
NUM_MACHINES 6
 2  1  0  0  3  0  1  6  2  3  7  0  5  3  0  4  6  2
 1  8  2  2  5  1  4 10  2  5 10  0  0 10  2  3  4  1
 2  5  2  3  4  0  5  8  2  0  9  1  1  1  2  4  7  1
 1  5  1  0  5  2  2  5  0  3  3  1  4  8  1  5  9  2
 2  9  2  1  3  2  4  5  2  5  4  0  0  3  0  3  1  2
 1  3  1  3  3  2  5  9  0  0 10  0  4  4  1  2  1  1

 0  0  1
 0  0  1
 1  0  0
```

B.2 MT10-TT

```
NUM_OPERATIONS 100
NUM_CLASSES 10
NUM_JOBS 10
NUM_MACHINES 10

 0 29  3  1 78  9  2  9  1  3 36  6  4 49  0  5 11  8  6 62  7  7 56  2  8 44  9  9 21  6
 0 43  6  2 90  3  4 75  8  9 11  8  3 69  8  1 28  5  6 46  3  5 46  9  7 72  9  8 30  7
 1 91  4  0 85  0  3 39  2  2 74  7  8 90  8  5 10  1  7 12  4  6 89  2  9 45  2  4 33  5
 1 81  2  2 95  9  0 71  9  4 99  6  6  9  4  8 52  3  7 85  8  3 98  4  9 22  3  5 43  0
 2 14  0  0  6  7  1 22  9  5 61  8  3 26  4  4 69  7  8 21  0  7 49  9  9 72  0  6 53  0
 2 84  5  1  2  7  5 52  3  3 95  9  8 48  9  9 72  1  0 47  3  6 65  4  4  6  4  7 25  8
 1 46  9  0 37  5  3 61  0  2 13  4  6 32  1  5 21  0  9 32  5  8 89  6  7 30  3  4 55  4
 2 31  4  0 86  0  1 46  8  5 74  2  4 32  1  6 88  4  8 19  8  9 48  8  7 36  4  3 79  9
 0 76  7  1 69  3  3 76  0  5 51  1  2 85  4  9 11  6  6 40  1  7 89  9  4 26  8  8 74  9
 1 85  0  0 13  2  2 61  9  6  7  7  8 64  3  9 76  3  5 47  7  3 52  0  4 90  8  7 45  3


 0 11 25 11  3  0  3 17 11 21
21  0 15 15 23 21 18 16  3  8
 7 18  0 12  3  7  7  4  0 24
20 24 17  0 15  1  4 23 11 18
18 15  4  3  0  6  1 15  3  9
17  3 20 24 10  0  5 13 12 20
10 12 10 25  9  7  0  9  2 17
 8 12 12  7  9 16 24  0 14 14
25  9  8  2 16 25  4 25  0  8
 1 16 16 22 19 22 10 10 21  0
```

B.3 MT20-TT

```
NUM_OPERATIONS 100
NUM_CLASSES 10
NUM_JOBS 20
NUM_MACHINES 5

 0 29  3  1  9  7  2 49  9  3 62  9  4 44  4
 0 43  7  1 75  9  3 69  3  2 46  0  4 72  0
 1 91  5  0 39  2  2 90  5  4 12  1  3 45  4
 1 81  2  0 71  3  4  9  2  2 85  5  3 22  6
 2 14  7  1 22  3  0 26  2  3 21  1  4 72  3
 2 84  8  1 52  7  4 48  6  0 47  4  3  6  0
 1 46  9  0 61  4  2 32  0  3 32  1  4 30  1
 2 31  4  1 46  9  0 32  4  3 19  4  4 36  4
 0 76  9  3 76  1  2 85  4  1 40  0  4 26  8
 1 85  4  2 61  9  0 64  3  3 47  3  4 90  5
 1 78  3  3 36  6  0 11  1  4 56  7  2 21  9
 2 90  3  0 11  8  1 28  9  3 46  7  4 30  9
 0 85  7  2 74  0  1 10  1  3 89  5  4 33  6
 2 95  8  0 99  6  1 52  0  3 98  7  4 43  3
 0  6  0  1 61  6  4 69  6  2 49  2  3 53  4
 1  2  9  0 95  8  3 72  2  4 65  8  2 25  6
 0 37  4  2 13  0  1 21  6  3 89  3  4 55  7
 0 86  5  1 74  9  4 88  5  2 48  3  3 79  7
 1 69  3  2 51  8  0 11  9  3 89  0  4 74  9
 0 13  3  1  7  1  2 76  2  3 52  9  4 45  5

 0 19 13 12 19  5  2 18  3 14
 1  0 20  4  6 21 22  0 20 11
 8 11  0  8  9  1  6  1 19 24
 8 24  3  0  5 10 13  5 10 18
 4  6 15  6  0  2 11  7 20  5
16 14  9 14  8  0  5 24 15 18
10 16  4 22 14 23  0  9  6 17
14 23 22  6  7 17 25  0 23 10
18  9 15 11  7  0 20 12  0  9
 6 11 21 16  9  8 21 16 12  0
```

B.4 ABZ5-TT

```
NUM_OPERATIONS 100
NUM_CLASSES 10
NUM_JOBS 10
NUM_MACHINES 10

 4 88  6  8 68  3  6 94  6  5 99  1  1 67  1  2 89  0  9 77  9  7 99  8  0 86  8  3 92  3
 5 72  5  3 50  7  6 69  2  4 75  4  2 94  9  8 66  4  0 92  4  1 82  3  7 94  0  9 63  8
 9 83  5  8 61  5  0 83  5  1 65  3  6 64  9  5 85  1  7 78  7  4 85  3  2 55  2  3 77  6
 7 94  9  2 68  0  1 61  0  4 99  3  3 54  3  6 75  4  5 66  0  0 76  7  9 63  0  8 67  6
 3 69  4  4 88  2  9 82  3  8 95  4  0 99  5  2 67  1  6 95  8  5 68  3  7 67  4  1 86  7
 1 99  7  4 81  5  5 64  7  6 66  6  8 80  1  2 80  8  7 69  3  9 62  3  3 79  3  0 88  9
 7 50  3  1 86  9  4 97  6  3 96  3  0 95  5  8 97  9  2 66  7  5 99  8  6 52  1  9 71  3
 4 98  2  6 73  1  3 82  0  2 51  0  1 71  8  5 94  8  7 85  0  0 62  3  8 95  9  9 79  4
 0 94  2  6 71  6  3 81  4  7 85  0  1 66  9  2 90  5  4 76  0  5 58  6  8 93  2  9 97  5
 3 50  4  0 59  5  1 82  9  8 67  4  7 56  3  9 96  2  6 58  1  4 81  9  5 59  1  2 96  5

 0 30 36 28 22  5 21 25  4 21
20  0 34 12 30 32  7 13 11  7
 0 21  0 31 36 20 28  4  2 12
16 36 27  0 33  6 33 33  7  9
20  6 18  0  0 26 22 20 32 10
21 18 36 22 18  0 32 17 15 17
11 19 32 26  4 30  0 10 18  1
 1  3 35 30 25 10 12  0 33 33
25 26 13 32  5  0  4 33  0 16
33 36 14 26 33  9 18 25 28  0
```

B.5 ABZ6-TT

```
NUM_OPERATIONS 100
NUM_CLASSES 10
NUM_JOBS 10
NUM_MACHINES 10

 7 62  9  8 24  1  5 25  4  3 84  4  4 47  0  6 38  5  2 82  5  0 93  6  9 24  1  1 66  1
 5 47  5  2 97  5  8 92  5  9 22  8  1 93  9  4 29  8  7 56  3  3 80  8  0 78  7  6 67  5
 1 45  4  7 46  1  6 22  2  2 26  9  9 38  0  0 69  5  4 40  6  3 33  5  8 75  4  5 96  9
 4 85  4  8 76  0  5 68  4  9 88  1  3 36  2  6 75  5  2 56  0  1 35  8  0 77  6  7 85  6
 8 60  4  9 20  9  7 25  0  3 63  0  4 81  3  0 52  1  1 30  2  5 98  2  6 54  5  2 86  0
 3 87  9  9 73  6  5 51  4  2 95  3  4 65  1  1 86  8  6 22  7  8 58  6  0 80  8  7 65  0
 5 81  9  2 53  4  7 57  3  6 71  5  9 81  8  0 43  3  4 26  0  8 54  4  3 58  1  1 69  8
 4 20  8  6 86  4  5 21  2  8 79  5  9 62  2  2 34  5  0 27  5  1 81  8  7 30  4  3 46  2
 9 68  8  6 66  9  5 98  9  8 86  5  7 66  3  0 56  4  3 82  9  1 95  7  4 47  1  2 78  8
 0 30  8  3 50  4  7 34  9  2 58  7  1 77  7  5 34  4  8 84  1  4 40  4  9 46  6  6 44  6

 0  2  9  3 19 12  4  7 24  7
 5  0  2 11  5  8  9  3 16 17
26  3  0 20 28  4  8 21  5 10
25  3  5  0  8  8 21  3  3 17
24  0 23  6  0 21 11 25  9 28
14 23 20 12 11  0  5  9 10 15
13 14 15  2 20 23  0 29 21  7
24 29 11 14 20 20 27  0 20 13
 9 28  4 16  6 20 15 29  0 12
15 25 23 21 22 25 19 22  3  0
```

B.6 ABZ7-TT

```
NUM_OPERATIONS 300
NUM_CLASSES 30
NUM_JOBS 20
NUM_MACHINES 15

 2 24  6  3 12 25  9 17 19  4 27 11  0 21 24  6 25 15  8 27  1  7 26 28  1 30  0  5 31  8 11 18 21 14 16  0 13 39 11 10 19  2 12 26  6
 6 30 24  3 15  0 12 20 19 11 19 18  1 24 18 13 15  7 10 28 14  2 36 26  5 26 20  7 15 27  0 11 23  8 23 29 14 20  3  9 26 15  4 28 28
 6 35 22  0 22 25 13 23 15  7 32 17  2 20 11  3 12 24 12 19 20 10 23 12  9 17  6  1 14 24  5 16 22 11 29 15  8 16 12  4 22  2 14 22 22
 9 20  5  6 29 24  1 19 19  7 14  1 12 33 10  4 30 28  0 32  5  5 21  0 11 29 11 10 24 16 14 25 28  2 29  8  3 13 14  8 20  6 13 18 20
11 23 19 13 20 12  1 28  2  6 32 26  7 16 24  5 18 12  8 24 13  9 23 23  3 24 19 10 34  1  2 24 23  0 24 27 14 28  1 12 15 19  4 18 28
 8 24 23 11 19 27 14 21 21  1 33 23  7 34 21  6 35 11  5 40 29 10 36 11  3 23  4  2 26 28  4 15 11  9 28 18 13 38 18 12 13 14  0 25 13
13 27 20  3 30 14  6 21 24  8 19 16 12 12  3  4 27  0  2 39 29  9 13 26 14 12  1  5 36 13 10 21 16 11 17 23  1 29 25  0 17 26  7 33 14
 5 27 27  4 19 16  6 29 19  9 20 21  3 21 18 10 40 19  8 14 26 14 39 28 13 39  8  2 27 28  1 36  4 12 12 22 11 37 22  7 22 25  0 13 14
13 32 18 11 29 13  8 24  0  3 27  2  5 40  7  4 21 21  9 26 18  0 27 25 14 27  1  6 16 22  2 21  3 10 13 29  7 28 26 12 28 25  1 32 25
12 35 24  1 11 20  5 39 22 14 18  8  7 23  5  0 34 22  3 24 18 13 11 10  8 30 15 11 31 25  4 15  2 10 15 15  2 28 24  9 26 21  6 33 15
10 28 10  5 37 28 12 29 27  1 31 17  7 25 20  8 13 29 14 14 20  4 20 28  3 27 18  9 25 15 13 31  3 11 14  1  6 25 13  2 39 15  0 36 23
 0 22  0 11 25 16  5 28 10 13 35 13  4 31 24  8 21  1  9 20 26 14 19  1  2 29  3  7 32  1 10 18 12  1 18  5  3 11 13 12 17 26  6 15 23
12 39 22  5 32 10  2 36 17  8 14 24  3 28 19 13 37  4  0 38  1  6 20 19  7 19 18 11 12  1 14 22 15  1 36 16  4 15 11  9 32 27 10 16 23
 8 28  3  1 29 25 14 40 27 12 23  1  4 34 14  5 33 17  6 27  0 10 17 17  0 20  6  7 28 11 11 21  7  2 21 11 13 20 27  9 33  7  3 27 22
 9 21 17 14 34 18  3 30  8 12 38 21  0 11  8 11 16 11  2 14  7  5 14 17  1 34  7  8 33 16  4 23 29 13 40 14 10 12  1  6 23  0  7 27  3
 9 13 25 14 40 16  7 36 25  4 17 13  0 13 25  5 33 24  8 25 21 13 24  1 10 23 14  3 36 15  2 29 17  1 18 29 11 13 25  6 33  7 12 13 22
 3 25 10  5 15 15  2 28  2 12 40 24  7 39 12  1 31 22  8 35 18  6 31 12 11 36  5  4 12 20 10 33  9 14 19  2  9 16 11 13 27 14  0 21 10
12 22 10 10 14 22  0 12 13  2 20  4  5 12 22  1 18 10 11 17 24  8 39 19 14 31  3  3 31 26  7 32 14  9 20  8 13 29 12  4 13 24  6 26 15
 5 18 26 10 30  4  7 38 28 14 22 14 13 15 19 11 20 28  9 16 23  3 17 11  1 12  1  2 13  3 12 40  0  6 17  8  8 30  4  4 38 17  0 13  0
 9 31 29  8 39 24 12 27 16  1 14 22  5 33 17  3 31 23 11 22 23 13 36  2  0 16 16  7 11 20 14 14  3  4 29  6  6 28 17  2 22  4 10 17  0

 0  3  6 11 10  6  7  4  5  8  1  4  2  5  7  4  5  8  7  2  0  7  4  5 11  8  6  5  7  3
 6  0  2  9  6 11  3 11  9  1  3  9 11  4 10  9  2  0  7 11  8  8  8  9  1  0  0  1  9 10
 8  0  0  3  4  6 11  1  4  4  1  3  5  1  4  0  1 10  6  2  1  2  4  1  5  9  0  0  1  3
 3  7  5  0  8  2  9  2 10  9  5  4  7  6 11  1  6  7 11  1  3  6  4  8  5  0  4  6  1  9
 0  5 10  2  0  0 12 11  6  2  6  9  7  4  0  2  8  8  0  0  9  3  9  3  4  9  8  4  1  9
 6  0  2  3 10  0  0  9  5  8  3  8  2 11 11  6  2  3  4  8  6  4 12  3  4  4  6  0  6  6
10  9 11  8 11  2  0  8  7  0 10 11  9 10  1  4  3  6  9  1  8  9  0  4  2  6  4 11  9  9
 1 11  6  9  2  3 12  0  7 10  5  0  2  1  0  9  4  4  6  8 11  5  8  6  4  7  7  2  8  4
 5  8  3  7  2  5  7  5  0  8  0  4  0  9 10  2  4  0  3  7 10  7  4 12  8  2 12  8 11  5
 0  4  4 11  3  7  9 10  1  0 11  9  9  4  8 11 12  9  0  4  8  7  9  1  7 11  8  6  4  8
 3  0  2  7  3  9 11  9  7  8  0  4  9  0  0  7 11  5  0  5 10  2  7  5  6  7  3  5  0  8
 1  9  2  3  1  4  2 11  6  7 10  0  8  8  5  7 11  5 10  8  9  7  9  5  2 12  2  8 10  1
 4  7  7  5  4  7 12  1  6  0  9 10  0 11  0  4  7  8  3  9 11 11  1  4  3  6  0  5 11  6
 6 11  6  7 11 10  5  8 11  2  5  4  9  0 11 11  0  7  6  9 10  7  3  6  8  6  0 11  4  2
 0  1  9  1  4  2  7  3  7  0  0  9 11  9  0  6  2  8  8  0  4  6 11 11  5  1  0  7 11  0
 3  1  0 11  1  7  1  1  0  7  3  0  3  1  5  0  8  4 10  5 10  3  6  1 11  5 11  1 10  5
 9 12  0  2  8  7  0  5  4  3  6  8  4  9 12 11  0  5  0  2  0  5  8  2  8  7  2 10 11 10
 0 10  7 11  1 10  3  0  2  0  6 12  7  3 11  4 10  0  4 11 10  4  6 12  1  5  4 11  0  8
 5  7 11 10  3  5  4  6 10  3  2  6  6 10  1  2  3  3  0  4  8  8  1 12 11  1  9  1  4  9
10  2  4  5  2  5  5  2  8 10  1  7  9  4  4 10 11 10  8  0 10  8  2  9  5  2  0 11 10  8
 7  4  9  5  0  9 10  9  7 10  6  2 11  4  1  5 10  7  2  0  0 11  5  8  5  9 11  0  0  9
 4  9  2 11  7  6  6  1  2  0 11  0  5 11  7  8  7  8 10 11  8  0  8  8  9  6 10  4  0 10
 6  3  6  9  5  5  5  4  0  0  5  6  7  9  3  8  1  9  9 11 11  8  0  0 11  2  1  1 11  9
 9  1 11  0  9  2  8  7 12  8  3  0  9 10  7  8 10 11  4  4  3  4  0  0 10  3 10  0  5  7
 8  0 11  6  3  3  2  0  5  1  5  9  2  1  4  5  7  8  7  1  8  5  0  9  0  3  3 10  3 10
10  2  6  6  1  7  9  8  8 10 10  4 11  7  4  8 10  5 11  8 11  2  0  5  1  0  1  0  1  0
 7 12 10  7  9  0  3  3  1  2  9  9  6  8  7  7  9  8 12  9  7 11  6  2  4  7  0  0  9  9
 5  9  1  0  0  8  7  5  4  3  2  1  0  0  8  7  2  7  5  2  2 10  2  1  0  3  1  0  7  2
 7 12  7  9 11  0  8  8  2  3  4  9  6  2 10 11  0  6  2 11  3  5  5  8  4 11  0  0  0  7
10  7  9  2  8  4  4  9  4  1  5  5  6  7  4  4  5  7  0  4 10  6  5  0  3  4  6  5  9  0
```

B.7 ABZ8-TT

```
NUM_OPERATIONS 300
NUM_CLASSES 30
NUM_JOBS 20
NUM_MACHINES 15

 0 19 25  9 33  8  2 32 21 13 18 20 10 39 17  8 34 24  6 25 19  4 36 28 11 40 17 12 33 11  1 31  4 14 30  0  3 34  1  5 26  3  7 13 29
 9 11 23 10 22 14 14 19  8  5 12 20  4 25 12  6 38  0  0 29 11  7 39  9 13 19 21 11 22 23  1 23 12  3 20 19  2 40 16 12 19  3  8 26 27
 3 25 24  8 17 17 11 24 15 13 40 26 10 32 21 14 16 21  5 39 17  9 19  2  0 24 16  1 39 20  4 17 17  2 35 21  7 38 11  6 20 23 12 31  1
14 22 21  3 36 12  2 34 16 12 17 23  4 30 10 13 12 18  1 13 11  6 25 22  9 12 27  7 18 12 10 31 21  0 39 19  5 40 20  8 26 12 11 37 21
12 32  2 14 15  5  1 35 12  7 13  8  8 32  7 11 23  5  6 22  1  4 21 14  0 38 14  2 38 26  3 40 29 10 31 17  5 11 25 13 37 28  9 16  9
10 23 12 12 38 25  8 11  6 14 27 28  9 11  5  6 25  6  5 14 11  4 12 16  2 27 28 11 26  4  7 29 11  3 28  1 13 21 14  0 20  5  1 30 28
 6 39 24  8 38 12  0 15 12 12 27  7 10 22  7  9 27 15  2 32 25  4 40 23  3 12 12 13 20 22 14 21 10 11 22 13  5 17  3  7 38  7  1 27 19
11 11  3 13 24 27 10 38  2  8 15 29  9 19 19 14 13 25  5 30  2  0 26 11  2 29 17  6 33 14 12 21 12  1 15 26  3 21 10  4 28  9  7 33 27
 8 20  3  6 17 11  5 26 17  3 34 28  9 23 23  0 16  1  2 18  2  4 35 13 12 24 23 10 16 16 11 26  2  7 12 10 14 13 14 13 27 23  1 19  2
 1 18 28  7 37  2 14 27 16  9 40 19  5 40  2  6 17  3  8 22 25  3 17  9 10 30  9  0 38 20  4 21 15 12 32 25 11 24  6 13 24 25  2 30  8
11 19  9  0 22 18 13 36 23  6 18 16  5 22 25  3 17 16 14 35  0 10 34 12  7 23 18  8 19 26  2 29  7  1 22 20 12 17 19  4 33  8  9 39 11
 6 32 22  3 22 29 12 24 13  5 13 25  4 13 29  1 11 14  0 11 22 13 25  6  8 13 16  2 15 18 10 33 11 11 17  4 14 16  2  9 38 15  7 24 25
14 16  5 13 16 15  1 37  3  8 25 19  2 26 10  3 11  9  9 34 10  4 14 21  0 20  3  6 36 27 12 12 15  5 29  7 10 25  5  7 32 25 11 12 10
 8 20 18 10 24 24 11 27  2  9 38 23  5 34  7 12 39 14  7 33 13  4 37 22  2 31 17 13 15 14 14 34  9  3 33 25  6 26 17  1 36 15  0 14 17
 8 31 27  0 17 29  9 13  6  1 21  5 10 17 23  7 19 22 13 14 18  3 40 26  5 32  4 11 25 27  2 34 11 14 23  2  6 13 12 12 40 15  4 26  6
 8 38 26 12 17 21  3 14 15 13 17 20  4 12 10  1 35 17  6 35  9  0 19  3 10 36  9  7 19  9  9 29  2  2 31  6  5 26 16 11 35  4 14 37 18
14 20 10  3 16 17  0 33 25 10 14  8  5 27 15  7 31  7  8 16  7  6 31  6 12 28  5  9 37 24  4 37 26  2 29 22 11 38  9  1 30  7 13 36 29
11 18  8  3 37 21 14 16  3  6 15 26  8 14  6 12 11  5 13 32 29  5 12  9  1 11  7 10 29 21  7 19 10  4 12 15  9 18 29  2 26  1  0 39 13
11 11 20  2 11 28 12 22 25  9 35  0 14 20 20  7 31 16  4 19 26  3 39 15  5 28 16  6 33 29 10 34  8  1 38  0  0 20 17 13 17 27  8 28 18
 2 12 15 12 25 23  5 23 20  8 21 22  6 27  4  9 30 11 14 23  2 11 39 20  3 26  4 13 34 28  7 17  8  1 24 26  4 12 17  0 19  4 10 36 28

 0 10  1  5  5 11  2  2  8  3  9  7  6 11  9  9  2  2 11  4  6  1 12  3  6  9  0  5  2 12
 5  0  5 11  7  7  7 10  3  9  7  8 11  4  3 12  9  6  5  7  2  4  3  2  4 10 12  1  4  6
 8  4  0  0  9  2  8  6  5 12  1  3  7  8  7  2  9  6 11  4  5  1 11 12  5  0  7  8  4  6
 6  4  5  0  5  9 11  7  6  4  1  5  8  1 11  1  7  7 10  1  1  9  9  7  8  3  2  9 12  4
12  7  3  3  0  1  4  7  0  1  4  2  1  1  0  7  2  1 12  0 12  8  6 11  4 10  8  6 11  6
 9  8  7  0 10  0  2  5  1 10  0  3 10 10  3  0  4  6  9  3  5  1  0  1 10 11  8  0  2  8
 6  0  0  6  6 12  0  2  9  7  9  0  0  5  4 12  2  5  6  0 11 10 11  0 12  4  5  6 11  2
 0  1 11  1  0  0  2  0  9 11  6  5  3  3 10  3  6  6  3  0  8 11  6  5 11  2 11  1 11  6
 5  1  1 11  8  0 12  4  0 11  9 10  4 11  2  8  1  7  6  0  8  0 12  6 10  3  9  8 11  9
 9  3 12  2  0  2 11  6 12  0  5  6  4 10  0  9 11  9  5  5  8  6  8  7  5  4  1  8  2  7
11  4  0  0  7  7  9 10  1 11  0  2  1  3  1  2  2  7  3  9  2 10  6  7 12  0  2  3  6 10
 3  2  6  5  5  9 11  1  1  4  9  0  5  4 11  2  8 11  8  1 10  8  5 10  8 10  3  6  9  3
 1  6  8  4  6 11 12  8 10  4 12  6  0  5  7  9  3  6  1  4  9  0 12 11  1  7  5 12  8  8
 5  4  2  2  4  0  9  7 11  2  2 10  8  0  6 10  6  3  7  3  7  5  5  7 11 11 12  7  9  0
 5  9 12  0  5 10  5  9  9  6  1  6  9  7  0 12 10  6  1  5  4  7  7  6  8  6  4 11 10  3
 7 12  2  2  8  8  3  7  9  7  5  4  7  0 10  0  4  2  4  9  6  8  4  4  3 11  3 12  9  1
 2  8 10  1  2  8 11  2  5  2  1  0  0 10  5  3  0  3  5  6 12  7 12  2  9  6  7 12 10  9
 9  8  2 11  0 10 12 11  5  5  2  1  4  8  4 12  3  0  6  8  3 11 10  4  2  3 10  8  4  2
 2  3  5  9  4 10  5  4  2  4  6  0  6  7 11  7  6  1  0  2 11  2  2  7 10 11  3  3  9  3
 0  8  3  4 10  5  0  2  5  5  1  0  2 11  3 12  4  0  1  0  1  0  3 10  9  2 10  3  1  7
 9 10  9  4  4  8 10  7  8  1  2  5  9  5  5 11 11  3  4  8  0  0  3 11  9  6  9 11  4  1
 6  1  6  2  5 10  4  6  0 12  6 12  8 10  4  7  5  1  5 12  7  0  7 10  9  4  0  3 11  6
10  2  7  5  2  9 11  8 10  9  7  0  1 10  6 10 12  0  1 10  3  8  0  2 11 12  0  1  1 12
 3  2  6  1  4  7 11 10 11 12  0  3  4  1 10  5  7  6  1  0  0 11  6  0  8  7  6  7  6  8
 4  5 11 12 10  4 11  9 11  6  5  0  0  5  3  3 11 11  3  4  8  4 10  4  0  8 12  1  3  7
 0  5  5 10 12  0  3 12  0 10  3  2  9  4  7  8 12  5  6  0  2 11  2  6  8  0  8 11  5  8
 7 11  7  1  7  8  3  3  8  8  6 11 10 11  8  0 11 12  0  2  9  1  9  8  4 12  0  4  6  3
 9  3 10 10  9  9  1  3  3  5  8 11  2 12  6  5  9  7  5  3 11 12  1  9  5  6  0  0 12 11
 2  9  0 11  7  7  7  3  9  6  2 11  5  3 11  5 10 12  9 11  5  0 11 11  2 10  4  8  0 11
12  9  0  7  7  6  9  1 10  7  8  1  2  6  6  7  2  9  7 12  7  1  9 10  8 11 10 12 10  0
```

B.8 ABZ9-TT

```
NUM_OPERATIONS 300
NUM_CLASSES 30
NUM_JOBS 20
NUM_MACHINES 15

 6 14 13  5 21  4  8 13 22  4 11 23  1 11 14 14 35 18 13 20 23 11 17 10 10 18 27 12 11 14  2 23 25  3 13 12  0 15 28  7 11 27  9 35 28
 1 35  3  5 31 13  0 13 22  3 26  6  6 14 17  9 17  9  7 38  3 12 20 23 10 19  2 13 12 13  8 16 15  4 34  3 11 15 23 14 12 22  2 14  2
 0 30 23  4 35 13  2 40 16 10 35 19  6 30 12 14 23 18  8 29 14 13 37 18  7 38  1  3 40 26  9 26  3 12 11 13  1 40 21 11 36  4  5 17  7
 7 40 23  5 18  6  4 12 27  8 23 27  0 23 22  9 14  6 13 16 24 12 14  3 10 23  6  3 12 10  6 16  4 14 32 19  1 40 20 11 25 17  2 29 11
 2 35 13  3 15  6 12 31 19 11 28  9  6 32  5  4 30 22 10 27 17  7 29 17  0 38 22 13 11 16  1 23  5 14 17  4  5 27 23  9 37  5  8 29 19
 5 33  0  3 33  5  6 19 19 12 40  5 10 19 13  0 33  7 13 26 21  2 31 10 11 28  9  7 36  7  4 38 29  1 21 18 14 25 10  9 40 27  8 35 24
13 25 19  0 32 22 11 33  1 12 18 28  4 32 11  6 28 14  5 15 10  3 35  7  9 14  4  2 34 14  7 23 20 10 32  4  1 17  9 14 26 23  8 19  5
 2 16 26 12 33  0  9 34 18 11 30  9 13 40 27  8 12  9 14 26 12  5 26 21  6 15  1  3 21  4  1 40  7  4 32  5  0 14  9  7 30  4 10 35 13
 2 17  8 10 16  7 14 20 23  6 24 13  8 26 18  3 36 17 12 22  8  0 14 11 13 11 18  9 20  4  7 23  5  1 29 24 11 23 22  4 15 21  5 40  9
 4 27 20  9 37  5  3 40 28 11 14 10 13 25 10  7 30  3  0 34 20  2 11 22  5 15 29 12 32  2  1 36  8 10 12 10 14 28 20  8 31  8  6 23 26
13 25  3  0 22 15  3 27  6  8 14 10  5 25 17  6 20 25 14 18  4  7 14 29  1 19  9  2 17 24  4 27 28  9 22 12 12 22  1 11 27  3 10 21  9
14 34  4 10 15 23  0 22  2  3 29 27 13 34 22  6 40 12  7 17 14  2 32  5 12 20 19  5 39 12  4 31 19 11 16 17  1 37 28  8 33 25  9 13 13
 6 12 28 12 27 25  4 17 13  2 24 29  8 11 22  5 19 17 14 11 27  3 17  2  9 25 19  1 11 18 11 31  8 13 33 28  7 31 24 10 12  6  0 22  8
 5 22 17 14 15 22  0 16 26  8 32 28  7 20 18  4 22  3  9 11 25 13 19  4  1 30  4 12 33 27  6 29  0 11 18 25  3 34  5 10 32 26  2 18 22
 5 27 23  3 26  7 10 28 16  6 37 29  4 18 23 12 12 18 11 11  7 13 26 16  7 27 26  9 40  6 14 19  2  1 24 14  2 18 17  0 12  7  8 34  4
 8 15 23  5 28 18  9 25 11  6 32 11  1 13  1  7 38 26 11 11  6  2 34 29  4 25 26  0 20 22 10 32  9  3 23 25 12 14 19 14 16 11 13 20 23
 1 15  4  4 13 13  8 37 17  3 14 18 10 22 11  5 24 19 12 26 15  7 22 16  9 34 19 14 22 26 11 19 24 13 32  5  0 29 11  2 13  9  6 35 12
 7 36 17  5 33 10 13 28 11  9 20  2 10 30 15  4 33 15 14 29  9  0 34  7  3 22  5 11 12  2  6 30 16  8 12 19  1 35 23  2 13  9 12 35  0
14 26  0 11 31 28  5 35  0  2 38 27 13 19 24 10 35 29  4 27 20  8 29 25  3 39 21  9 13  7  6 14 10  7 26 10  0 17  4  1 22 18 12 15 13
 1 36  8  7 34 19 11 33  2  8 17  7 14 38 14  6 39 24  5 16 21  3 27 12 13 29 27  2 16 16  0 16  4  4 19 18  9 40 18 12 35 13 10 39  5

 0  9  1  7  6  7  1  3  6  9  1  2  6  1 10  8  2  1  3 11  3  2  6  3  8  3  8 11  1  1
 0  0  0  0  5  7  5  8  5  0 11  5 10  7  5  7  4  3  4  2  3  8  6  1  0  2  1 12  5  2
 2  4  0  7  9  5  8 10  9 11 11  7 10  2  5  1  1  9 10  9  8  4  0  9  6  7  8 12  1  1
10 11  4  0  0  7  7 11 11  3  1  3  9  2  7 10  4  6  8 11  1  1  8  2  1  8  2  5  9  6
 9  0  5  6  0 11 10 10 10  1  9  1 10  6  8  8  9  4  3  2  1  8  0  1  8  9  1  2  2  6
 7  7  8  8 11  0  4  0  6 11  5  5  6  4  7  2  5 11  0  1  4  7  7 10  3 10  1  0  1  7
 4  3  3 11  3  3  0  4  8  4  3 11  5  9  0 10  3  4  2 10  7 10  2  6  4  7 10  5  8  5
 0  6  3  9  7  3  7  0  9 11 11  5  1 11  0  8  3  9  5 10  6  3  1 12 11  2  2  2  8  4
 4  0  8 10  1  6  8  7  0  1 10  7  6 12  3  8  7 11 10  1 12  2  9  6 11  2  9  9  9  3
 8  4  3 10  4  7  1  4  0  0  6  8  4  1  3  4  7  6  2  8  6  5  1  4  4  6 12  3  4  4
 7 10  0  9  4  7 11  7  3  9  0  5  4  0 12  8  7  9  1  5  5  0  4 11  8  1  0  8 12  7
 6  7  9  3  9  2  3  9 10  7  1  0 11 11  9  5  2  5  8  0 11  7  5  6  9  9 11  4  3  5
 7 11  7  6 11  6  7  0  2  2  7  6  0  9  1  0  4  5  0  0 11  8 11  2  3  1  3  3 11  6
10 10  4  9  3  0  0  4  9  9  9  7  7  0  6  1  3  2  2  9 11  3  0  3  4  1 11  8  4  5
 6  6  5  2  5  9  3  2  6  9 12  3  0  2  0  8  7  1  8 11  9  5  7  0  1  8 10  1  2 12
10  9  7 10  2  7 11  8  2 12  7  7  1  7  6  0  1  0  0  8  2  4  7  6  9  5  4  9 10  0
 5  3  0  1  0  7  9  5 10  2  8  6 11 10  4  4  0  2  2  7  2 10 11 10  1  1  0  7  3  7
 5  6  2  7  1  7  1  7  4 10  7 10 10  6  9  8  1  0 10  9  7  1  8  1  9  0  5  6  1  5
 4  4  2  7 11  4  1  1  7 11  4 11  1  9  4  1  9  4  0  4 12  4  2  9  2  5  5  9  4  5
 2  4  2  0  4 11  6  6  4  2  1 11 10  6  4  8 11  1  0  0 11  4  6 10  6  6  6  3  4  2
 4  8  8  8  7  9  4  3  1 11  0  0  3  7  2  8 11  9 11  0  0  9  2 11  8  2  2  1  5  6
 8  0 10  8  8  1  1  5  9 10  6  5  4  3 11  5 10  7  4  9  7  0  3  8 12  7  8 11  9 11
 1  6  7  2  6  0  0  2  9  7  5 10  7  8  1  7  5  7  2  5  2  4  0  9  1  9  4  4  1  9
 0  4  7  9  1  5  5  8 10  7  2  3  2  4 11  7  9 11  4  9  2  6  9  0  7  5  2 10  2  7
 5  0  3  1  2  3  6  4 10  6  0  8  6  7  0  7  7  8  4  3 11  8 11 11  0  8  7  3  7  6
 3 10  5 10  2  5  9  3 10  0  4  5  1 10  4 10  7  0  6 10  3  3 10  9  5  0  6  9 10  6
 4  3  0  2  3  1  0  2  9 10 12  3  2  0 10  6  2 11 11  4  4  5  5  6  5  6  0  6 11  9
 5  1  7  8  4  2  2  0  5  6  1  8  6  0 10  0  0  7  6  9  9  6  2  1  8  0 10  0  6  8
10  2  8  9  7  6 11  9  0  3  0  1  9  8  5  5 10  8 11  1  4  9  0  9 11  6 12  4  0 10
 3  1  2  8  7  9  2  4  6  3  1  2  3  5 10  2  2  9  7  1  2  5  6  8  1  0  0 11  9  0
```