Minimizing the Makespan for Unrelated Parallel Machines

Y. Guo[3], A. Lim[1], B. Rodrigues[2] and L. Yang[3]
[1]Department of IEEM, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
[2]School of Business, Singapore Management University, 469 Bukit Timah Road, Singapore 259756
[3]School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543

June 1, 2003

Abstract

In this paper, we study the unrelated parallel machine problem for minimizing the makespan, which is **NP**-hard. We used Simulated Annealing (SA) and Tabu Search (TS) with Neighborhood Search (NS) based on the structure of the problem. We also used a modified SA algorithm, which gives better results than the traditional SA and developed an effective heuristic for the problem: Squeaky Wheel Optimization (SWO) hybrid with TS. Experimental results average 2.52% from the lower bound and are within acceptable timescales improving current best results for the problem.

Key Words: makespan, unrelated parallel machines, simulated annealing, tabu search, squeaky wheel optimization

1 Introduction

In this work we study the problem in which n jobs are scheduled on m unrelated parallel machines without preemption: Suppose we have n jobs, and each job is to be assigned to exactly one of the m machines. Job j ($j = 1, \cdots, n$) becomes available for processing at time zero and requires a processing times p_{ij} if job j is assigned to machine i ($i = 1, \cdots, m$). The objective is to schedule jobs so that the makespan, C_{max}, which is the finishing time of the latest finished job, is minimized.

Garey and Johnson (1979) showed that the problem is **NP**-hard even for a special case of identical machines, $R||C_{max}$, where the processing time of each job does not depend on the machine to which it is assigned. This notation is from Lawler et al [13]. R indicates that the problem is for unrelated parallel machines. No notations between the two — means the problem is non-preemption, no set up time for jobs, and the release times for all the jobs are the same. C_{max} indicates that our objective is to minimize the makespan. Van de Velde (1993) proposed an exact algorithm using a branch and bound technique. However, computational requirements are prohibitive for problems with more than 5 machines and 50 jobs. Martello et al. (1996) developed lower bounds for this problem based on Lagrangian relaxation and additive techniques.

Because the problem is **NP**-hard, effort has been directed into designing fast and efficient approximation algorithms with good performance bounds. Glass et al. (1994) studied local

search heuristics for this problem and Piersma and Dijk (1996) provided an efficient search technique which we adapt and use in this work.

This paper is organized as follows. In section 2, we develop a lower bound for the minimum makespan problem which we will use in experiments. In section 3, we discuss two local search heuristic methods: SA with Neighborhood Search (SA+NS) and TS with Neighborhood Search (TS+NS), and a modified SA+NS. In section 4, we introduce another heuristic method - Squeaky Wheel Optimization (SWO) - for this problem, which is used with local search. In section 5, we combine SWO with our previous SA+NS and TS+NS. In section 6, we compare our experimental results among different heuristic methods developed.

2 Preliminaries

In this section, we describe a loose lower bound C_1 and then a tighter lower bound C_2 which we use to compare the quality of various heuristic methods in this work.

2.1 Lower Bound

Define, for $1 \leq j \leq n$, $D_j = min_{i=1,\dots,m}\{p_{ij}\}$ and $C_1 = \frac{\sum_{j=1}^{n} D_j}{n}$. Obviously C_1 is a lower bound for the problem since it sums possible minimum processing times of each job under the assumption that all machines are busy processing some job. The lower bound C_1 is loose as we can expect. Next, we define job j's second smallest processing efficiency, $\xi(j)$ by: $\xi(j) = \frac{\text{job } j\text{'s second minimum processing time}}{\text{job } j\text{'s smallest processing time}}$. In order to improve the lower bound C_1, we iteratively calculate a tighter lower bound C_2 as described in **Algorithm 1**.

Algorithm 1 The algorithm to find lower bound C_2

$C_3 \leftarrow Minimum\ total\ processing\ time/Number\ of\ machines$
$C_2 \leftarrow C_3$
while $C_2 \leq C_3$ **do**
  Remove jobs with the lowest $\xi(j)$ values with preemption from machines with completion time greater than C_2 to their second fastest machines.
  $C_3 \leftarrow New\ total\ processing\ time/Number\ of\ machines$
  $C_2 = C_2 + 1$
end while
$C_2 = C_2 - 1$

The idea here is to increase the total processing time from the possible minimum value step by step. The value of the new total processing time over number of machines, namely C_3 in the algorithm, is intuitively a lower bound. The above algorithm ensures that each time we increase the total processing time, the increase is the minimum possible. While $C_2 \leq C_3$, we can say that C_2 remains a lower bound since C_3 is the possible minimum makespan at that point of time. Only when $C_2 > C_3$, we can not tell that C_2 is still a lower bound. Thus we take the last C_2 value as the lower bound.

3 SA+NS, TS+NS and a modified SA+NS

3.1 Neighborhood Generating Mechanisms

We develop a local search method here where two operations are used to generate local search neighborhoods. These operations are the:

- *Move Operation:* Reassigning one job from a machine with maximum completion time to another machine
- *Swap Operation:* Swap one job from a machine with maximum completion time with one job from another machine

These two operations are illustrated in Figure 1 and Figure 2.
In Figure 1, we have 14 jobs and 4 machines. The number in each rectangular box is the label of a job. The length of the rectangular box represents the processing time of the job on that machine. Machine 1 and 2 have the maximum completion time. The move operation reassigns job 1 from machine 2 to machine 4.
In Figure 2, the swap operation swaps job 1 on machine 2 with job 6 on machine 4.

3.2 Simulated Annealing with Neighborhood Search

Simulated annealing is an optimization metaheuristic based on the cooling properties of solids (Kirkpatrick, Gelatt and Vecchi, 1983) and is a local search algorithm capable of escaping local optima using hill-climbing moves. Here, we implement an SA approach to the problem.

3.2.1 Implementation of Neighborhood Generation in SA

- *The Move Operation*
  We investigate all jobs assigned to the machines that have maximum completion time among all machines. If we can assign one job to other machines we may be able to decrease the makespan. For SA, we use $C(S)$, to denote the makespan of solution S and we accept a move with probability $e^{-\Delta(C)/T}$, where Δ represents the difference in costs and T the temperature, in the SA process. Since we use SA, we need to make moves as random as possible. If we always move a job from one of the machines with maximum completion time to the machine with the smallest label, i.e. machine 1, when as T is large in the beginning, machines with small labels will have larger makespan because jobs from machines with maximum completion time are always moved to these machines, and machines with large labels will have small makespan. Besides, we must take into considerations the order of machines we assign jobs to in the move operation when T is very small, to guarantee a possible downhill move. To overcome these problems, we use the following steps of control in the implementation:

  - Step 1:
    Check all machines to find those machines with maximum completion time. We may find more than one machines.

- Step 2:
  Randomly reorder the labels of those machines found in step 1 and record their new labels.
- Step 3:
  Randomly reorder the labels of machines other than those found in step 1 and record their new labels.
- Step 4:
  Move a job from the machine with smallest label in step 2 to a machine with smallest label in step 3.

These four steps assure the SA process has moves that are sufficiently random.

- *The Swap Operation*

  Choose a machine from those machines that have the maximum completion time, and randomly choose another machine. Check the jobs assigned to these two machines to see whether we can swap a pair of jobs between the two machines. We can use the same probability function used in the Move operation to decide whether to accept a swap. In addition, the steps of control are also used in the swap operation.

3.3 Modified SA+NS

We have experimentally found it possible to improve the quality and running time by modifying the existing SA heuristic; this is done as follows: instead of selecting one neighbor, we select m acceptable neighbors and use the best one.

This modification decreases the effect of temperature. After many experiments, we find that this change can decrease running time without any degradation in solution quality.
Figure 3 gives the pseudocode of Modified SA+NS algorithm:

3.4 TS+NS

Tabu Search is a metaheuristic developed by Glover (Glover 1986, 1989, 1990, 1997). The basic strategy is to avoid becoming trapped in cycles using memory techniques by forbidding or penalizing moves, in the next iteration, which take the solution to points in the solution space previously visited (hence "tabu").

Selected attributes that occur in solutions recently visited are labeled tabu-active and solutions that contain tabu-active elements, or particular combinations of these attributes are tabu.

There are four types of memory structures in Tabu Search: Recency-based memory; Frequency-based memory; Quality-based memory and Influence-based memory. Using these memories, we have two important components of TS which are intensification and diversification.

Intensification strategy is based on modifying choice rules so as to encourage move combinations and solution features historically found to be good by thoroughly searching part of solution space.

Diversification strategy encourages the search process to examine unvisited regions and to generate solutions that differ in various significant ways from those seen before and can be achieved by radically moving to another part of the solution space.

To avoid retracing the steps used, the method records recent n steps' moves in one or more Tabu lists. Any tabu move is forbidden. However, if a solution we get from a move is the current best solution we have, we accept this move even if it is tabu, which is an improved-best aspiration criterion.

3.4.1 Control flow of TS

There are four parts for the TS implementation for our problem; these are: Controller, Intensification and Diversification, Restart and TS. Each time, Controller passes the elite result list containing all elite results found so far to Intensification and Diversification. Intensification and Diversification then analyzes those elite results to get some constraints for restarting. Restart receives and follows these constraints to get one intensification solution and one diversification solution which will be returned to Controller and be used as initial solutions in TS in the next iteration. TS searches the neighbors of these two initial solutions and generates a list of elite solutions. Within TS, some randomly chosen elite results are used as initial solutions to start with, and update the elite solution list when appropriate. Later, TS returns this elite result list to Controller which needs to start the new round of processes if stop criterion is not met. Figure 4 shows the flow of TS method.

- *Controller*

  Controller guides TS to find a good solution more efficiently, and passes the information between Intensification and Diversification component and TS component.

- *Intensification and Diversification*

  These are two important components of TS which use some memory-based techniques to guide search direction.

  Intensification: We find common elements. Here, a common element is the same job assigned on the same machine of elite results. To get an intensification result, we continue to use these common elements. For other elements that are not the same, Restart is used.

  Diversification: We find common elements of those elite results, and for those common elements, we assign the jobs to a different machine where it can run fastest.

- *Restart*

  The component receives the partial solution and those unassigned jobs from the Intensification and Diversification component, and assign those jobs to the machines. Here, we use the following two rules while assigning the jobs:

  Rule 1: Assign the job to the machine such that the resulting makespan is the smallest among different assignments.

  Rule 2: If there is more than one assignment that will result in the smallest makespan, adopt one of the assignments where the job on the machine runs fastest among these assignments.

  Rule 1 is natural since when we assign jobs to machines, we want to minimize the makespan; Rule 2 is to ensure the efficiency of a job because though different assignments may result in the same makespan, the time to process the newly-assigned job should be taken as the selection criterion.

- *Tabu Search*

  The most important component in Tabu Search is the neighborhood generating mechanisms. We need to search all the neighbors of the current solution and choose the best non-tabu neighbor as the new solution. We also use the improved-best aspiration criterion explained above. We store all elite solutions within pre-defined tolerance to the current best solution and use these elite results as initial solutions to run TS for the next iteration.

  In the implementation, when a better solution is found, we refined the elite solution list. Since we have improved the best solution, previous elite results may be outside the tolerance used and thus removed from the elite solution list. We randomly select an elite solution from this new elite solution list, and perform the new TS iteration. If we cannot find a better solution in a prescribed number of iterations, we restart using the diversification strategy.

  The control flow for TS is shown in Figure 5.

3.5 Time Complexity for Neighborhood Search in SA+NS and TS+NS:

Let n and m be the number of jobs and machines respectively.

1. In SA+NS, for the two random reorderings, the time complexity is clearly $O(m)$.

2. For Move Operation: Considering the worst case, all the machines have the same completion time, so each job will have $m-1$ possible moves. So totally there are $n(m-1)$ possibilities. Hence the time complexity is $O(nm)$

3. For Swap Operation: Considering the worst case, still all the machines have the same complete time, so each job will have $(n-x)$ (here, x is the number of jobs on the same machine) possible change pairs. Again for the worse case, $x = 1$. So totally $n(n-1)$ possibilities. Hence the time complexity is $O(n^2)$ under the ordinary situations when $m = O(n)$.

It shows that the time complexity of finding a neighbor is $O(n^2)$.

4 Squeaky Wheel Optimization

4.1 Overview

SWO is a general metaheuristic approach to optimization problems developed by Joslin and Clements (1999). We can describe it as an iteratively greedy approach. The core of this algorithm is a Construct/Analyze/Prioritize cycle which continues until some stop criteria is met or some acceptable solution is found.

In the Constructor component, a solution is constructed using a greedy algorithm. The priorities used in the greedy algorithm are assigned in the Prioritizer component according to the blame factor of the solution. That solution is then analyzed to find the elements that are "trouble makers". The priorities of the troublemakers are then increased, causing the Constructor to deal with them earlier in the next iteration.

4.1.1 The Three Components of SWO

- **Constructor**: For a sequence of problem elements, the Constructor generates a solution using greedy algorithm according to the order given by Prioritizer.

- **Analyzer**: In the solution constructed by Constructor, for each element, the Analyzer gives it a numeric blame factor according to the problem's constraints.
  By analyzing a solution, we can often identify the elements of the solution that work well, and also the elements that work poorly.

- **Prioritizer**: Using the blame factor, the Prioritizer modifies the previous sequence of the problem elements. The elements that receive a higher blame factor are moved towards the front of the sequence, and thus, will be dealt with earlier in the next iteration.

4.2 Problem Transformation

4.2.1 Transformation within polynomial time

We transform the original problem to a new problem so that we can employ the SWO heuristic. This transformation runs in polynomial time. The idea is to try to guess a makespan C'_{max}, and then try to assign the jobs to machines without overshooting the makespan C'_{max} using heuristic methods - SWO - here. If we cannot find a feasible makespan which is not greater than C'_{max} using SWO, we increase our guess of C'_{max}.

We divide the problem into four main components similar to the TS heuristic, namely the Controller, Initializer, Restart and the SWO approach:

1. The Controller invokes the Initializer for an initial solution and then uses this solution to make a guess of the bound for C_{max}, i.e. C'_{max}
2. The Controller sends the solution got from Initializer together with the bound and C'_{max} to the SWO component
3. The SWO component generates a solution and sends the solution to the Controller.
4. The Controller analyzes the solution got from the SWO component to tighten the bound of C_{max}, and decides whether to restart.

The control flow is shown in Figure 7.

4.2.2 Implementation

- *Controller*

  This component will guide SWO to find a sequence of better solutions by tightening the bound of C_{max}. It first gets the Initializer to build a solution, and then analyzes the solution to get an upper bound U and a lower bound L. We set the guess C'_{max} for C_{max} as $(U + L)/2$, and then pass this C'_{max} to SWO as an initial solution. Each time SWO

generates a solution, the Controller analyzes the solution following Rule 3 to Rule 5:
Rule3: If we find an arrangement such that the makespan is less than U, we set U to be that makespan.
Rule4: If the SWO component cannot improve the solution in N (prescribed) iterations, we take the guess C'_{max} as too small, and set L to be C'_{max} and C'_{max} to be $(U + L)/2$.
Rule5: If the difference between L and U is less than our defined tolerance, we stop and take U as our final result.

- *Initializer*

  Initializer is used for generating an initial solution, which puts each job onto the machine that processes it fastest. This initial solution is taken as the upper bound U of the makespan. If we add up all the jobs' process time and divide it by the number of machines, we can get the lower bound (Lemma 1).

- *Restart*

  If we cannot improve the current best solution after N iterations, we consider the $C'_{\max}$ we are using is not feasible. So we need to increase the bound of C_{max} and use the newly-guessed $C'_{\max}$ to construct a new solution.

4.3 Implementation of SWO

As C'_{max} is a guess for the makespan C_{max}, our aim in SWO is to assign all the jobs to the machines so that the makespan is no greater than C'_{max}.

Each time, we use priority to select a job and greedy strategy to find a machine for the job. In the Analyzer component, we analyze the result using rules we defined to give a numeric blame factor to each job.

We describe the implementation of SWO in terms of three main components.

- *Constructor*

  The Constructor generates a solution by assigning the jobs one at a time in the order they occur in the priority sequence. We choose a job with the highest priority in the job list given by Prioritizer and assign the job to the machine using the following rule:
  Rule6: If there are some machines such that after assigning the job to them, the makespan is still no greater than $C'_{\max}$, we will select the machine where the job has the fastest processing time; if no matter which machine is selected, the makespan will exceed C'_{max}, we mark this job as unassigned and apply Rule 1 and Rule 2 to assign the job later.

- *Analyzer*

  The Analyzer will assign a numeric blame factor to each job. The blame assigned to a job is its "excess cost" (the difference between its actual processing time and its shortest processing time). The value of the blame factor of job j depends on the value $b(j) =$ (current process time of $j -$ minimum process time of j)/maximum process time of j.

- *Prioritizer*

  Once the blame factor has been assigned, Prioritizer modifies the previous sequence of jobs by moving jobs with higher priority forward in the sequence.

For each job j ($1 \leq j \leq n$), we define the total process time $TPT(j) = \sum_i p_{ij}$ and the maximum total processing time to be $TPT_{max} = \max_{j=1,\ldots n}(TPT(j))$. We then define the *Ratio*, $R(j)$, of a job j by $R(j) := \frac{TPT(j)}{TPT_{max}}$ and an additional rule:
Rule7: Choose to assign the job that has the largest *Ratio* first.

Using the combination of the blame factor, which is derived from Rule 7 and the job's previous priority, we define the priority of job j in a new iteration of SWO by: *Priority* $(j) = \alpha.b(j) + \beta.R(j) + \gamma.b'(j)$, where α, β, γ are constants which are determined from experimental results which give best performance of the SWO heuristic and $b'(j)$ is the current priority assigned to job j.

Figure 8 is the pseudocode of Prioritizer:

4.4 Implementation of SWO with Iterative Improvement Local Search

In order avoid the intrinsic problem that SWO may become trapped in cycles, we need to restart periodically. As we know, SWO can have large changes in both prioritizer space and solution space: A small change in the sequence of elements generated by the Prioritizer can correspond to a large change in the solution generated by the Constructor. Hence, by restarting periodically, we hope to change the solution generated by Constructor greatly and thus avoid small cycles.
To reduce the drawbacks of SWO and Local Search, we use SWO and Local Search together. For every iteration, after Constructor constructs a solution, we pass this solution to a Local Search as an initial solution, fro which SA will return a local minimum solution. Here, we use an Iterative Improvement Local Search which has the constraint that every move must be a downhill move, which means the result of every move must cause the maximum makespan to be unchanged or be reduced. In our implementation, we used the two neighborhood operations in SA and set initial temperature to be zero.

The modified SWO cycle is shown as Figure 12:

5 SWO with SA+NS and SWO with TS+NS

SWO with Iterative Improvement Local Search gives better results than SWO or Iterative Improvement Local Search alone as determined from experimental results. This led to combining SA and TS with SWO. For the combined SWO heuristic, we make use of the Analyzer and Prioritizer to analyze the best solution from SA or TS, and construct a new solution as the initial solution for the next iteration of SA or TS.

5.1 SWO with SA+NS

For each result generated by Constructor, we use it as an initial solution to run SA. After meeting some stop criteria, we return the best result found by SA in this iteration to Analyzer. After many tests, we found that SWO with SA did not have obvious advantage over SA. This is possibly due to the fact that the effect of SWO is negated by the randomness present in the initial part of SA when temperature is large.

5.2 SWO with TS+NS

We consider this method as one which uses TS to refine the result obtained from Constructor of SWO. This approach can also be viewed as using SWO to guide the restart in TS instead of using intensification and diversification strategies. This method give us the best solution in both quality and running time among all the methods, as found from experiments.

Figure 11 is the pseudocode of SWO with TS+NS:

6 Experimental Results

We run the test cases on an Intel Pentium 2 Processor (300 MHz) with SD-RAM 192MB. Figure 11 shows the results of different heuristic methods.

In the table, m is the number of machines, n is the number of jobs, and Time is CPU time based on second. To measure the quality of the results from different heuristics, we define relative error by $\Delta =$ (Solution - C_2)/C_2.

6.1 Test Case Generation

We use process times, p_{ij} ($1 \leq i \leq m$; $1 \leq j \leq n$), which are is uniformly distributed within the interval [10, 100].

6.2 Comparison between Heuristics

Based on the quality of the five heuristic methods, we found from experiments that the heuristics performed in the following order, with SWO as the worst performer and SWO with TS as the best: SWO, SWO with Iterative improvement, SA, SWO with SA, TS, Modified SA, SWO with TS. For running times, the fastest was SWO and the slowest was SWO with SA: SWO, SWO with Iterative improvement, SWO with TS, TS, Modified SA, SA, SWO with SA. As a result, taking the result quality as a major measure of the heuristics, SWO with TS+NS gives us the best results. Hence, we propose the SWO with TS+NS heuristic as the solution to our unrelated parallel machine problem.

6.3 Comparisons of the Results with Previous Work

We compared our results with van de Velde's results (column "vdV" in Figure 14) and Martello et al.'s results (column "MST" in Figure 14). Martello et al. developed a lower bound based on Lagrangian relaxations and other techniques which is different from the iterative-generated lower bound used here. The numerical values in column MST in Figure 14 are relative to that lower bound which we include here for completeness.

For comparison purposes, we generate small test cases such as test cases of 2 machines with 60 jobs, 3 machines with 40 jobs and 5 machines with 30 jobs. We use standard Branch and Bound (B&B) algorithm to get the optimal solution for these test cases (Refer to Figure 11). We found that our SA+NS, TS+NS, SWO with Iterative Improvement Local Search, SWO with SA+NS and SWO with TS+NS obtain optimal solutions for these test cases and perform better than MST and vdV.

Another point to notice is the tightness of the lower bound. From Figure 11, we see that when optimal results are available, the gap between the optimal result and lower bound can be

as large as 7%. Our best heuristic results (SWO with TS+NS) give the optimal results for all small test cases where B&B is applicable. For large test cases, SWO with TS+NS give good results as the gap between the heuristic and the lower bound ranges from 0.1 to 7 percent for most cases and an exceptional 18 percent for a test case with 20 machines and 50 jobs. The average error of SWO with TS+NS with large test cases is 2.52%.

7 Conclusions

The parallel machine scheduling problem is of importance to industry. In this paper, we studied the unrelated parallel machine problem for minimizing the makespan which is **NP**-hard. In a new approach, we used Simulated Annealing (SA) and Tabu Search (TS) with Neighborhood Search which was developed for this problem. We also used a modified SA algorithm, which gives better results than the traditional SA and developed an effective heuristic for the problem: Squeaky Wheel Optimization (SWO) hybred with TS. Performance improved significantly using the latter. Experimental results are good with solutions coming in with an average of 2.52% of the lower bound for large test cases and within acceptable timescales which improve on results achieved by others.

References

[1] Garey A.M and Johnson D.S, (1979) Computers and intractability: A Guide to the theory of NP-completeness, Freeman, San Francisco.

[2] Glass, C.A., Potts, C.N. and Shade, P. (1994) Unrelated Parallel Machine Scheduling Using Local Search. Mathematical Computer Modeling, Vol.20, No. 2, 1994, pp. 41-52.

[3] Glover, F. (1986) Future Paths for Integer Programming and Links to Artificial Intelligence. Computers Operations Research, Vol.13, pp. 533-549

[4] Glover, F. (1989) Tabu Search Part I. ORSA Journal on Computing Vol.1, pp.190-206

[5] Glover, F. (1990) Tabu Search Part II. ORSA Journal on Computing Vol.2, pp.4-32

[6] Glover, F. and Laguna, M. (1997) Tabu Search, Kluwer Academic Publisher

[7] Johnson, David S., Aragon, Cecilia R., McGeoch, Lyle A. and Catherin Schevon Optimization by Simulated Annealing: An Experimental Evaluation. Operations Research, Vol. 37, Issue 6, pp. 865-892.

[8] Joslin, David E. and Clements, David P (1999) "Squeaky Wheel" Optimization. Journal of Artificial Intelligence Research, Vol. 10, pp. 353 - 373.

[9] Kirkpatrick, S., Gelatt, C. and Vecchi, M. (1983) Optimization by Simulated Annealing. Science Vol. 220(4598): pp. 671-680

[10] Piersma, N. and Dijk, W.Van, (1996) A Local Search Heuristic for Unrelated Parallel Machine Scheduling with Efficient Neighborhood Search. Mathematical Computer Modelling, Vol. 24, No. 9, pp. 11-19.

[11] Martello, M., Soumis, F. and Toth, P, (1996) Exact and approximation algorithms for makespan minimization on unrelated parallel machines. Discrete Applied Mathematics, Vol. 75, 169-188.

[12] van de Velde S.L. (1993), Duality-based algorithms for scheduling unrelated parallel machines, ORSA Journal of Computing, Vol.5, pp. 192-205.

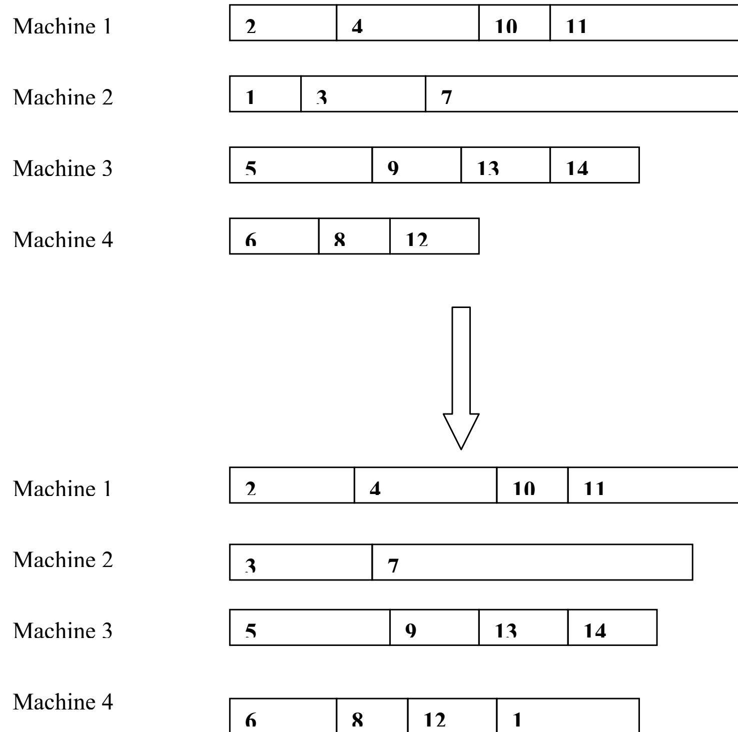

Figure 1: *Move Operation*

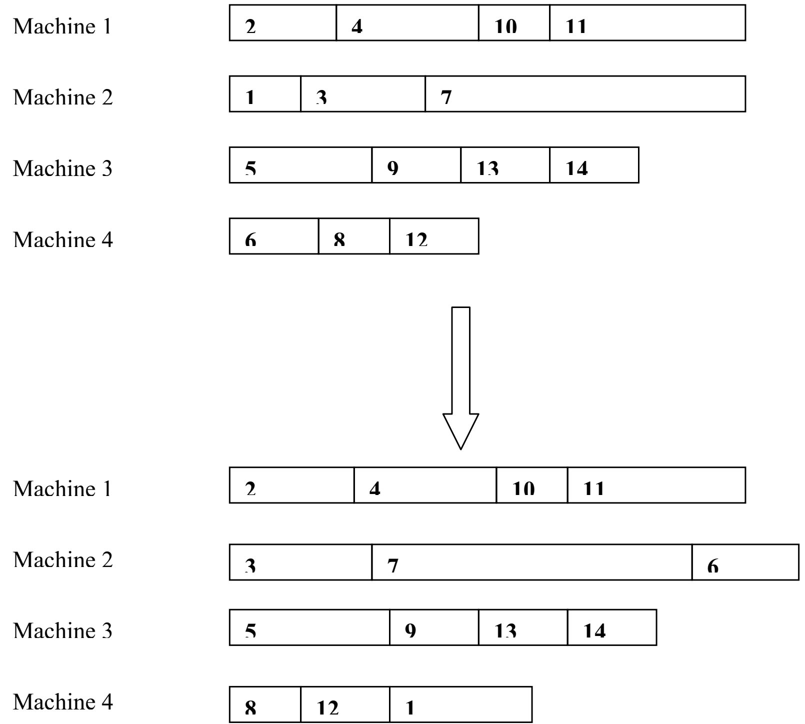

Figure 2: *Swap Operation*

ALGORITHM MODIFIED_SA

1. Generate an initial schedule S_0 and temperature T_0
2. Set the initial best schedule $S=S_0$
3. Compute cost of S_0: $C(S_0) = C(S)$
4. Set the temperature $T = T_0$
5. While stop criterion is not satisfied do:

5.1 WHILE have not got m acceptable results DO

5.1.1 Select a random neighbor S_1 to the current schedule, ($S_1 \in N(S_0)$)

5.1.2 Set $\Delta(C)=C(S_1)-C(S_0)$

5.1.3 If ($\Delta(C)\leq 0$) (downhill move)

5.1.3.1 Accept this solution

5.1.4 If Δ (C) > 0(uphill move)

5.1.4.1 Choose a random number r uniformly from [0, 1]

5.1.4.2 If $r<e^{-\Delta(C)/T}$, accept this solution

5.2 Select the best solution among these m neighbors as an initial solution of the next iteration

5.3 Every n iterations, update (or reduce) temperature T

6. Return the schedule S

Figure 3: *Modified SA+NS*

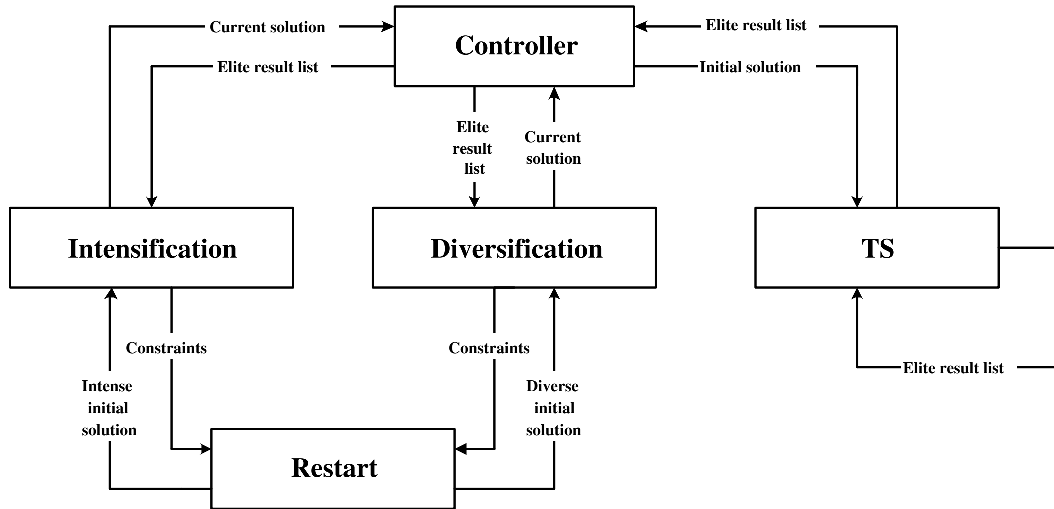

Figure 4: *Control flow of TS method*

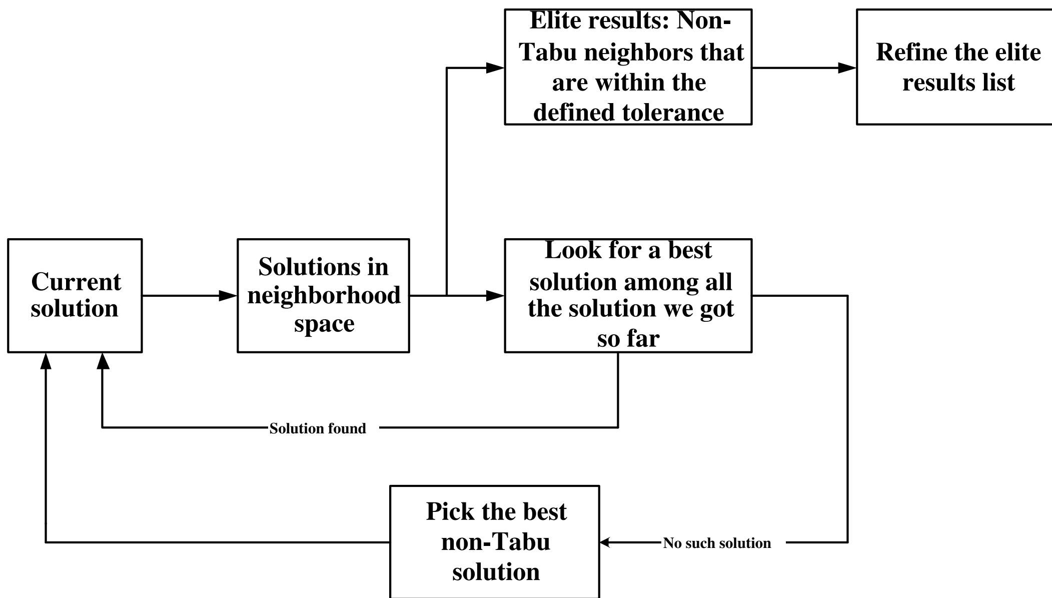

Figure 5: *Control flow for TS*

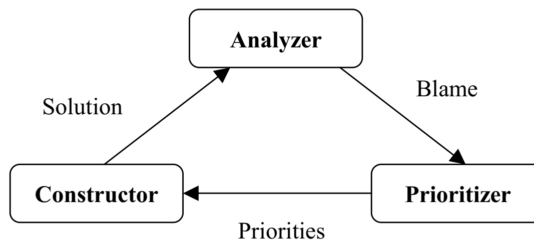

Figure 6: *SWO Construct/Analyze/Prioritize cycle*

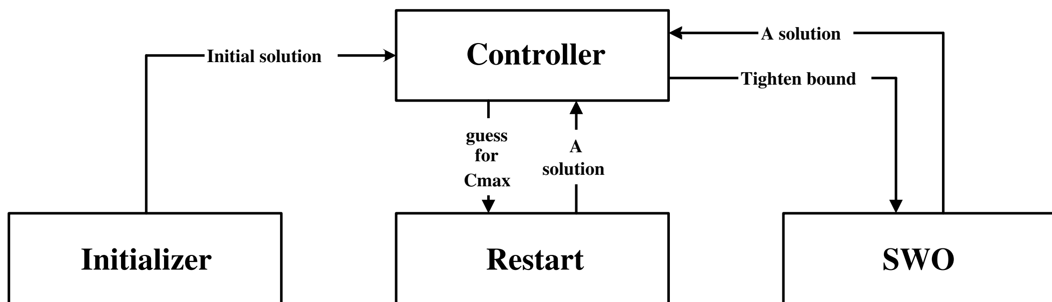

Figure 7: *Control flow of the method using SWO*

INPUT: Job list *J* and current job blame list *b'*
OUTPUT: job priority queue *Jq*
ALGORITHM PRIORITIZER (List *J*, List *b'*)

1. FOR each job in *J* DO
1.1 Set Priority (*j*) = α*b(j) + β* R(*j*) + γ * *b'*(*j*)
1.2 Insert this job into the priority queue *Jq* based on Priority (*j*)
2. return *Jq*

Figure 8: *Pseudocode of Prioritizer*

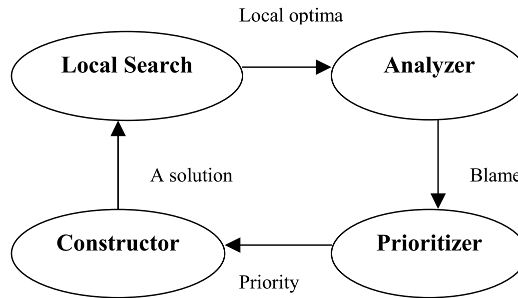

Figure 9: *SWO cycle with Local Search*

ALGORITHM SWO_TS ()

1. WHILE result has not been improved for N* iterations DO
1.1 Use the rules defined in SWO, Constructor generates a solution
1.2 Record this solution as an elite solution
1.3 WHILE result has not been improved for M iterations DO
1.3.1 Randomly select an elite solution
1.3.2 Use this solution as an initial solution to run Tabu Search
1.3.2.1 Record down all the elite solutions, which are those within some defined tolerance with the best result found so far
1.3.3 Refine the elite solutions (If the result has been improved, remove those results outside the tolerance)
1.4 Send the best solution to Analyzer and Prioritizer of SWO

*: N and M are constants defined by the program

Figure 10: *Pseudocode of SWO with TS+NS*

m	n	SWO		SA+NS		SWO with GL		Modified SA+NS		TS+NS		SWO with TS+NS		MST**		vdV		Optimal Solution (Branch & Bound)
		Time	Δ*	Time	Δ	Time	Δ	Time	Δ	Time	Δ	Time	Δ	Time	Δ	Time	Δ	Δ
2	20	0.01	0	0.1	0	0.01	0	0.1	0	0.01	0	0.01	0	1	0.0	1	0.0	1.5
	30	0.01	0	0.1	0	0.01	0	0.1	0	0.1	0	0.1	0	1	0.0	1	0.1	1.0
	40	0.05	0	0.1	0	0.05	0	0.1	0	0.2	0	0.2	0	1	0.1	1	0.2	0.7
	50	0.1	0	0.3	0	0.05	0	0.3	0	0.3	0	0.3	0	1	0.1	1	0.3	0.4
	60	0.1	0.1	1	0	0.1	0	1	0	1	0	0.5	0	1	0.0	1	0.2	0.2
	80	0.2	0.2	10	0.15	0.1	0.15	10	0.15	1	0.15	1	0.15	1	0.0	1	0.1	-
	100	0.3	0.2	20	0.1	0.5	0.1	20	0.1	5	0.15	1	0.1	2	0.0	1	2.3	-
	200	2	0.1	500	0.03	1	0.08	300	0.03	40	0.08	5	0.03	7	0.0	3	0.1	-
3	20	0.1	0	0.1	0	0.1	0	0.1	0	0.1	0	0.1	0	1	0.0	1	1.0	3.0
	30	0.1	0	1	0	0.1	0	1	0	0.2	0	0.2	0	1	0.1	1	1.2	1.5
	40	0.1	0	10	0	0.1	0	10	0	0.8	0	0.3	0	1	0.0	1	1.0	1.0
	50	0.1	0.2	20	0	0.1	0	10	0	1	0	0.4	0	2	0.2	1	1.1	0.9
	60	0.2	1	20	0.8	0.2	0.8	15	0.7	1	0.8	1	0.7	2	0.1	1	0.5	-
	80	0.2	0.8	20	0.5	0.3	0.6	15	0.5	1	0.5	1	0.5	2	0.1	3	0.7	-
	100	0.3	0.8	40	0.5	1	0.6	30	0.5	15	0.5	1	0.5	4	0.1	6	0.6	-
	200	2	0.4	600	0.15	2	0.2	200	0.1	60	0.1	30	0.1	11	0.1	40	> 0.5	-
5	20	0.1	0	2	0	0.1	0	3	0	0.1	0	0.1	0	1	0.6	1	5.4	7.0
	30	0.1	1.2	3	0	0.1	0	10	0	0.1	0	0.1	0	1	0.4	2	4.4	5.0
	40	0.2	5	4	3.5	0.1	3.5	20	3.5	0.2	3.5	0.2	3.5	2	0.1	12	3.8	-
	50	0.1	5	5	3	0.2	4.5	20	3	0.5	3	1	3	2	0.2	16	4.1	-
	60	0.3	3	10	1.5	0.3	2.5	30	1.5	1	1.5	1	1.5	5	0.3	33	2.7	-
	80	0.8	3	80	1.1	0.3	2.2	80	1	2	1	1	1	5	0.2	96	> 1.9	-
	100	1	2	100	1.1	1	1.8	80	1	5	1.1	3	1	8	0.4	87	> 2.2	-
	200	2	1	600	0.5	2	0.5	300	0.4	600	0.5	80	0.4	32	0.3	-	-	-
10	50	0.1	10	15	9	0.1	9.5	30	8.8	1	9	2	7	3	1.7	-	-	-
	100	0.1	8	30	5	0.1	6.0	60	4.0	8	3.5	8	3.5	5	1.3	-	-	-
	200	0.1	4	200	2.0	0.8	3.5	200	1.4	50	1.7	50	1.4	4	1.0	-	-	-
	500	1	3.0	350	1.2	1	1.2	300	1.0	1000	1.2	500	1.0	18	0.4	-	-	-
	1000	1	1.5	1200	0.6	1	0.8	1000	0.4	4000	0.6	1000	0.4	75	0.3	-	-	-
20	50	0.1	20	20	20	0.1	20	50	20	2	20	1	18	8	1.4	-	-	-
	100	0.1	11	100	10	0.1	10	80	8.5	10	8.5	5	8.5	13	2.0	-	-	-
	200	0.1	8	200	6.0	0.5	8.0	60	5.0	40	4.0	50	4.0	8	2.0	-	-	-
	500	1	4	300	2.6	2	3	500	1.8	100	1.8	500	1.8	28	1.1	-	-	-
	1000	2	2.7	1000	1.0	1	2	2000	0.7	500	1.5	2000	0.7	86	0.6	-	-	-

*Here Δ = (Solution we get from heuristic – lower bound)/lower bound * 100%
**MST uses a different lower bound from ours. The relative error is computed using his own lower bound.
The rightmost column is the optimal result from Branch and Bound if available. For testcases optimal result is available, the relative error Δ is compared with the

Figure 11: *Comparison of results between different heuristics and MST, vdV for* $R||C_{max}$