RC23384 (W0410-126) October 20, 2004
Mathematics

IBM Research Report

Allocation Problem with Sequencing in Steel Industry

Richa Agarwal, Jayant Kalagnanam, Chandra Reddy
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Allocation Problem with Sequencing in Steel Industry

Richa Agarwal, Jayant Kalagnanam, and Chandra Reddy
IBM T.J. Watson Research Center
Yorktown, NY 10541
{ragarwa, jayant, creddy}@us.ibm.com,

October 27, 2004

1 Introduction

Allocation problems with sequencing side constraints (APSSC) arise as a kernel optimization problem in production planning and operations scheduling in the steel industry. For example, continuous casting of slabs requires that the designed slabs are allocated to cast templates based on grade and associated capacity. However, the group of slabs allocated to a cast also need to be sequenced based on width transition constraints. Similary many of the finishing line processes require a similar kernel function. The hot strip mill is an operation that follows the caster where slabs are rolled into thin coils. The life of the roll is typically specified in terms of the the total weight (and other considerations) and this consitutes the size of a campaign for each roll. In addition, to the size of the campaign, the slabs have to be sequenced based on width and thickness transitions and need to be arranged in a coffin pattern based on the width. Many of these problems have nice mathematical structure and are closely related to some well known combinatorial problems such as the Travelling Salesman Problem (TSP), the Vehicle Routing Problem(VRP) and the time window related variations (TSPTW and VRPTW respectively) of these. However, a common structure common to all these is that the sequencing problems are asymmetric with different costs for sequencing 2 slabs depending on the direction. Often, only one of the two directions may be allowed (the cost cost is infinite) which leads to problem instances that have adjacency graphs that are not complete.

The focus of this report is twofold. The first is to visit the literature to study and benchmark some of the current algorithms that have been developed for asymmetric sequencing problems. The second is to study the hot strip mill scheduling problem in the steel industry and develop algorithms that are composed based on some of these kernel optimization functions for asymmetric sequencing. The report also studies the computataional performance of the proposed algorithms on randomly generatede instances of the hot strip mill scheduling mill problem with a size that would be appropriate for very large steel mills (in terms of capacity).

2 Literature Review

To get an understanding of the current state of TSP, VRP and some related problems we did a literature review. These problems are provably NP-hard thus it makes sense to develop heuristic (rather than exact) algorithms for them. For each of these problems we now report some of the construction and local improvement heuristics available in literature.

Let n represents the number of cities in our TSP. We will use vertex and city interchangeably to refer to the same thing.

2.1 TSP

Simple tour construction heuristics for the TSP involve nearest neighbor algorithm (always go to the next nearest un-visited vertex) , greedy algorithm(choose the least weight edge which does not create subtour) and the Clarke and Wright savings heuristic [9]. These algorithms are $O(n^2)$ and take less than 10 minutes on a million vertex problem. On randomly generated instances there results are usually within 20-15% of the optimum value. Christofides algorithm [10] which assumes triangle inequality for distance matrix is based on matching and minimum spanning tree problem. This algorithm provides a performance guarantee of 1.5 over the optimum.

Various local improvement heuristics have been studided for TSP based on the k-opt moves. In a k-opt move k links are broken and k different links are added to yield a route with better cost. Various implementations of 2-opt [14] and 3-opt [23] have reported results that are within 4.9% and 3% respectively of the optimum. Many meta-heuristics based on tabu search, genetic algorithms and simmulated annealing have also been applied to improve on TSP solution. The champion algorithms for this problem however are the various flavors of the Lin-Kernighan algorithm. The vanilla version of the algorithm [24] reports results within 1.5% of the optimum (within 45 minutes on a million vertex problem) whereas the more advanced versions [25] (based on parallelization, metaheuristic frameworks etc) report results that are within 0.5% of the optimum value.

The most commonly used lower bound for TSP is the Held Karp lower bound of [20], [21] which is based on the linear programming relaxation of the TSP.

2.2 VRP

The simplest construction heuristics for VRP are the Clarke and Wright [9] savings heuristic and the sweep heuristic of Gillet and Miller [18]. These heuristics take a few seconds on the standard test bed of Christofides et.al. [11] and are in about 6-8% of the best available results. Fisher and Jaikumar [15] proposed a heuristic based on generalized assignment problem which takes upto 4100 seconds to give results that are on an average 3.29 % above the best reported with Bramel and Simchi Levi [8] optimization for seed. The parallel savings heuristic of Altinkemer and Gavish [2] which tries to merge multiple clusters of nodes at each iteration takes 21 to 1150 seconds (depending on problem size) and their results matched the best available for some instance of the standard test bed.

Most of the successful metaheuristics for this problem are based on tabu search framework. The adaptive memory search of Rochat and Taillard [29] which is a general technique for any local seach reports the best available results in literature but the computational time for this approach are very high.

2.3 VRPTW

Solomon [30] enhanced various tour building heuristic as Nearest Neighbor, Clarke and Wright, Insertion and Sweep, that are available for TSP and VRP to VRPTW. The novelty of the approach consists in incorporating not only the distance but also the time dimension in the heuristic process through modified cost structure. They report that though the Nearest Neighbor is fastest it is the worst in terms of solution quality. Insertion heuristic performs the best across the instances generated by them.

The adaptive memory procedure of Rochat and Taillard [29] which utilizes probablistic diversification and intensifictaion and the Unified tabu search procedure of Cordeau et al [13] can both solve the VRPTW problem. Both of these algorithms give good result if the primary objective in VRPTW is to minimize the route length. Whereas, if the primary objective is to minimize the number of vehicles used then the two stage heuristics of Homberger et al [22] and Bent et al [7] are quite competitive. Both of these approaches try to solve the problem in two stages where the first stage tries to minimize the number of vehicles used

and the next stage minimizes the travel costs.

3 Asymmetric Travelling Salesman Problem

The main difference between the problems occuring in Steel industry and TSP/VRP is that the cost matrix is asymmetric in industry problems. TSP with asymmetric distance matrix i.e. where the distance from vertex i to vertex j is not same as the distance from j to i is refered to as the asymmetric travelling salesman problem or the ATSP. Johnson et al. [26] present an overview of various construction and local search heuristics for the ATSP. Balas and Simonetti [5] for a very restricted version of the TSP (asymmetric and symmetric) give a linear time dynamic programming based algorithm. They consider the TSP where a feasible tour is one in which vertex i precedes vertex j whenever $j \geq i + k$ in the initial ordering. They extend this result to general case where k is replaced by vertex specific integers k_j, for TSPTW and TSP with target times.

Since the ATSP is NP hard we do not have the luxury of comparing the performance of various heuristic algorithms against the optimal solution for large instances. The most commonly studied lower bound for ATSP is the Assignment Problem (AP) lower bound. This lower bound can be calculated by treating the distance matrix as representing the edge weights for a complete (symmetric) bipartite graph on $2n$ vertices and computing the minimum cost perfect matching for this graph. Clearly, the matching corresponds to a minimum-cost cover of the vertices by vertex disjoint directed simple cycles. Since, an optimal tour is the minimum cost cover of the vertices by a single directed simple cycle, the solution of the assignment problem provides a lower bound for the ATSP.

Johnson et al [26] have suggested another lower bound for ATSP which is based on the Held Karp lower bound for TSP. They transform the ATSP to TSP and then use the publicly available code **Concorde** of Applegate et al [3] to compute the Held Karp lower bound of TSP. To transform the ATSP to TSP they use the standard transformation that creates for each vertex i three vertices i_1 , i_2 and i_3. The distances $c[i_1, i_2] = c[i_2, i_3] = 0$, whereas distance $c[i_3, j_1] = c(i, j)$ for all pairs $i \neq j$; all other distance are 0.

Many construction heuristics for ATSP [28], [16], [19], known as cycle cover heuritsics, are based on assignment problem lower bound and essentially they try to patch the solution of the AP in different ways to get a solution for the ATSP. Among local search heurictics 3-opt and an extension of the classical Lin-Kernighan algorithm by Kanellakis and Papadimitriou [27] are available in the literature.

3.1 Implementation

We implemented various heuristics for ATSP in C++ using many of the combinatorial and geometric data types and algorithms from LEDA (Library of Efficient Data types and Algorithms) in our implementation. In all our impelmentations we view the problem as complete directed graph on n vertices.

3.1.1 Lower Bounds

We compute the Assignment Problem lower bound by generating a complete symmetric bipartite graph of $2n$ vertices using the original cost matrix for assigning weights. We then use the LEDA's built in function to compute the minimum weight matching. The cost of matching is reported as the lower bound.

3.1.2 Construction Heuristics

We implemented the following construction heuristics for the asymmetric TSP.

1. Nearest Neighbor. We start with a random vertex and then successively go to the nearest as yet unvisited vertex until the length of the tour becomes n.

2. Greedy. We sort the arcs in increasing order of their length using LEDA's list data structure. We then keep picking the smallest weight edge if it is eligible (i.e. does not make subtours and does not cause in and out degree to exceed one) until the number of picked edges become n.

3. Clarke and Wright. This is derived from a general vehicle routing algorithm. We choose a random hub vertex and create pseudo tours in which we return to the hub node after each visit to another vertex. We then define *savings* to be the amount by which the tour will be shortened if we went directly from one vertex to another, bypassing the hub. We then greedily go thru the non hub vertices in non-increasing order of savings, so long we do not get a subtour finally arriving at a tour that passes thru every vertex.

4. Patch Heuritisc. We first find the cycle cover using perfect matching on the bipartite graph as in the Assignment problem lower bound. Then using the Karp and Steele [28] heuristic we repeatedly select the two cycles containing the most vertices and patch them until we obtain a complete tour. To implement this we maintain the cycles in reverse order of the number of vertices they have in LEDA's priority queue data structure. i.e. The cycle with most number of vertices is at the top of the list.

   All heuristics return the final tour as an array on vertices.

3.1.3 Local Improvement Heuristics

To improve on the solution obatined by various construction heuristics we implemented 3-opt heuristic. In this heuristic we delete three edges thus breaking the tour into three paths and then reconnect those paths. Since the reversal of a path changes the lengths of all the arcs in the path and is difficult to evaluate, we reconnect the path in a way which does not change the orientation of any path. Note that for this reason an even-opt move in general is not good for improving on an ATSP solution. We used the neighbor lists of Lin and Kernighan [24] and two level tree structure of [17] (to be described in the next section) to perform the 3-opt move efficiently.

3.1.4 Data Structures

To have an efficient implementation of the various algoirthms we make use of the following data structures.

1. Neighbor Lists. We maintain for each node a sorted list of its k nearest neighbors using LEDA's sorted list data structure. This data structure allows us to quickly identify the allowable candidates for the 3-opt move.

2. Two Level Trees. Fredman et al [17] present a couple of data structures for tour representation. We pick the two level tree structure for tour representation in the 3-opt algorithm. As indicated in [17] this data structure has a $O(\sqrt(n))$ complexity for performing a 3-opt move, after such a move has been identified. Since we consider only rearrangement of vertices in the tour as a result of the 3-opt move (and no reversal of any subpath) we replace the original "flip" operation by an operation that can perform this 3-opt move directly and efficiently, though the worst case complexity still remains the same. The lower level of the two level tree representation consists of $O(\sqrt(n))$ consecutive subsegments (of the tour) of approximately equal length in a doubly linked list. The upper level contains a doubly linked list of parents of these segments. All members of the segments in the lower level contain a pointer to the corresponding parent node in the upper level. We refer the reader to [17] for the details of the data structure and present here the changes that we made to it.

   Let us say we wish to perform 3-opt move involving vertices $(v_1, v_2, v_3, v_4, v_5, v_6)$. Consider figure 1. As a result of the 3-opt move the tour changes from $L_1L_2L_3$ to $L_1L_3L_2$. To perform this move we perform the "separate" operation to separate segments contaning v_1 and v_2 if they are not already in

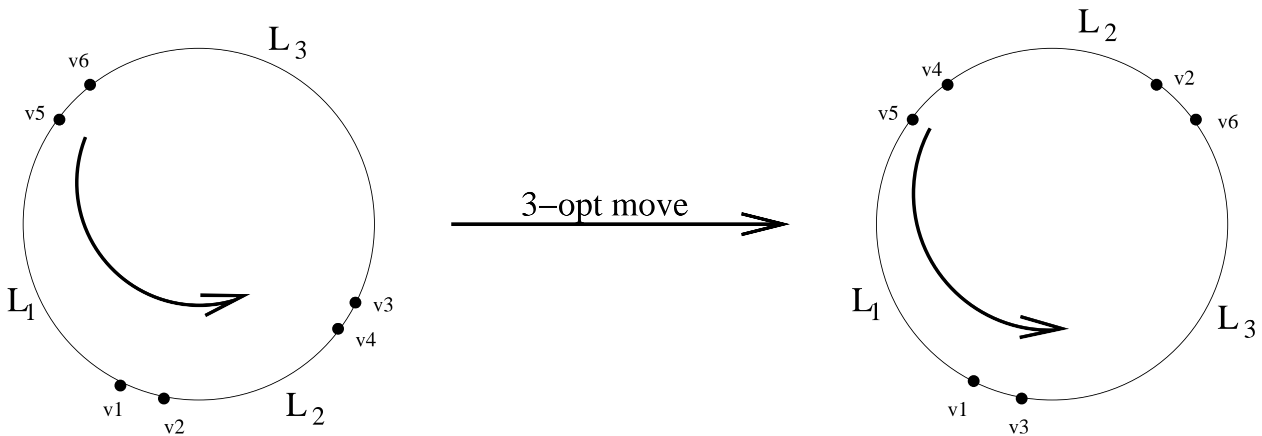


Figure 1: Effect of 3-opt move with no path reversals

different segments in the lower level of the tree. We do the same thing with v_3, v_4 and v_5, v_6. The 3-opt move can now be performed by changing a constant number of pointers. But, as a result of the 3-opt move the number of parents rises from $\sqrt(n)$ and the parent ids become corrupt. At the end of each iteration we update the ids of each parent. After every few iteration we check if the number of parents have gone above $2\sqrt(n)$ and do the rebalancing of the tree if required. To rebalance the tree we merge the small segments together. Merging two segments is an $O(\sqrt{n})$ operation but in worst case we might need to merge segments for all the parents. Thus the rebalancing operation is actually $O(n)$.

3.1.5 Results

We now present our results for the construction heuristics and 3-opt heuristic for the asymmetric travelling salesman problem. Our experiments were performed on a 256 MB PIII machine with 598MHz processor and with -O2 optimization for the gcc compiler. All times are in micro seconds and all results are averages over 5 randomly generated instances. Weights on edges were choosen uniformly from [0, 100].

Results in table 1 are reported on complete graphs. For 3-opt we used a neighbor list of size 10 and nearest neighbor algorithm was used to obtain an initial solution. For 3-opt we make use of two level tree data structure. As indicated earlier the most expensive operation on this data structure is the rebalancing operation. On an average 6 rebalances of the tree were required for a problem of size 100 and 13 for a problem of size 1000. *iters* column in table 1 reflects the number of 3-opt operations performed.

As evident from table 1 patch heuristic reports the best results for randomly generated ATSP instances but it also takes the most time. We used LEDA's implementation of the Hungarian algorithm for finding matching in a bipartite graph (used for computing the lower bound and for the patch heuristic). This is an $O(|V| * |A|)$ heuristic where $|V|$ represents number of vertices and $|A|$ represents number of edges in the graph under consideration.

We study the impact of density of graph on patch heuristic in table 2. LEDA was used to generate random graphs of various densities. In table 2 'Time' column in the Graph Generation reports the time taken to generate the graph and '% dev' columnn in Patch heuristic reports the % deviation of the solution obtained by patch heuristic from the assignment problem based lower bound. Clearly as graph becomes more dense it takes longer for the patch heuristic to construct the solution. Note that there is no specific relation between density of the graph and % deviation of the solution obtained by patch heuristic from the lower bound.

Data	Nearest Neighbor		3-opt			Greedy		Clarke-Wright		AP-Patch		AP-Lower Bound	
#Cities	Cost	Time	#iters	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time
10	286.37	71.40	1	253.07	789.80	205.60	574	210.79	621	187.04	1845.80	179.51	1462.20
100	509.66	4016.20	35	234.58	8.7E+4	453.26	1.0E+5	677.13	1.0E+4	182.69	1.6E+5	166.95	1.5E+5
1000	686.93	4.2E+5	149	306.71	2.3E+7	659.27	1.9E+7	1966.24	4.6E+7	170.44	4.2E+7	164.98	3.4E+7
2000	720.50	3.67E+6	196	337.5	1.49E+8	724.17	5.85E+8	2841.25	4.55E+8	166.73	4.53E+8	159.78	4.50E+8

Table 1: Result of various construction heuristics and 3-opt for ATSP(time in microseconds)

Data	50% Arcs				75% Arcs				100% Arcs			
#Cities/	Graph Generation		Patch Heuristic		Graph Generation		Patch Heuristic		Graph Generation		Patch Heuristic	
#Nodes	#Arcs	Time	% dev	Time	#Arcs	Time	% dev	Time	#Arcs	Time	% dev	Time
10	45	655.4	12.2	1411	67	768	10.39	1651	90	986	26.5	1894
100	4950	3.6E+4	86.39	9.0E+4	7425	62273	102.116	1.3E+5	9900	3.4E+5	89.48	1.9E+5
1000	499500	6.2E+6	109.13	3.0E+7	7429250	1.1E+7	84.1	4.6E+7	999000	6.6E+7	93.69	8.7E+7

Table 2: Peformance of patch heuristic on graphs of different densities (time in microseconds)

4 ATSP with time window and color constraints

To get closer to the steel industry problems we studied the asymmetric TSP with time window and color constraints. Each vertex has a release date, a due date, a processing time and a color associated with it. Each color has a maximum and minimum length of run associated with it and a feasible solution should have runs of same color within these ranges. Both the time window constraint and the color constraint are considered as soft constraints. Apart from the cost matrix associated with distance between verticies we have a cost matrix associated with transition from one color to another.

We modified the nearest neighbor and Clarke and Wright heuristic to accomodate the time window and color constraints in the ATSP. Our objective is to minimize the makespan of the schedule keeping the number of infeasibilites with respect to time window and minimum color length constraints as low as possible. If a schedule violates the minimum run length constraint at a point then it incurs a penalty proportional to the amount by which the constraint is violated. This is counted in the make span of the schedule. Also make span includes the transition cost from one color to another in the schedule.

Addition of a same color node in the schedule is always considered feasible. Let us suppose that the run of the current color is of length $curr_len$ and $s = \lceil \frac{curr_len}{max_run} \rceil$, where max_run denotes the maximum allowed run length of the current color type. Then clearly the addition of a node of different color is feasible if $s.min_run \leq curr_len$, where min_run denotes the minimum allowed run length of the current color type.

The notion of nearness in these algorithms combines various aspects thru a weighted cost structure. We define

$$cost_{(i,j)} = \alpha * c_{(i,j)} + \beta * increase\,in\,make_span\; + \gamma * urgency_{(i,j)}$$

here $c_{(i,j)}$ is the cost between nodes i and j. $increase\,in\,make_span$ is the total increase in make span of the schedule if we schedule node j after node i. If the sum of make span of schedule till node i, transition cost of color (if any) and the cost of violation of minimum run length (if any) is less than the release date of node j then increase in make span will be

$$release_date[j] + processing_time[j] - make\;span\;till\;node\;i$$

otherwise it will be

$$processing_time[j] + cost\;of\;run\;violations\;(if\;any)\; + transition\;cost(if\;any)$$

urgency captures the idea that if a job has an early due date then it should be scheduled early. It is simply calculated as:

$$due_date[j] - make\;span\;till\;node\;j$$

We also implemented a modified version of the simple nearest neighbor algorithm where we allow the algorithm to lookahead before scheduling a node.

5 Hot Strip Mill Problem

The next problem that we considered is the Hot Strip Mill (HSM)problem that occurs in the steel industry. For detailed description of the problem one can refer to [12], [32] and [4].

5.1 Data Generation

To study this problem we generate 5000 slabs. Each slab has a grade high, medium or low. Among the slabs that we generate 40% have high grade, 20% have medium grade and the rest 40% are of low grade.

Also each slab has a width, uniformly distributed from 1000 to 2500, a weight uniformly distributed from 20 to 50 tons and a thickness uniformly distributed from 5 to 50. Each slab has an earliest start time(EST) uniformly distributed from 0 to 24 hrs (we discretized time to 1 minute intervals and thus selected EST for each slab uniformly from 0 to 1440) and a latest start time (LST) which has a triangle distribution from 0 to 8 hrs (or from 0 to 480 minutes) where 0 is most heavily weighted and 8 is least weighted. Each slab also has a processing time triangle distributed from 1 to 3 minutes where 1 is most heavily weighted.

We wish to generate five campaigns each of weight no less than 30000 tons such that no slab of high grade is assgined after approximately 12000 tons and no slab of high or medium grade is assigned after approaximately 18000 tons. Two slabs can be adjacent in a campaign only if the difference in their width and thickness is less than 100 and 5 respectively. Also we wish that the profile of the campaign is such that on an average we follow a wide to narrow profile of slabs and slabs are processed with in their time window as much as possible. Keeping all these constraints in mind we wish to minimize the total time taken to process a campaign.

5.2 Construction Algorithms

In this section we present algorithms that we developed and implemented for the HSM problem. To construct an initial solution we used a heuristic based on Nearest Neighbor approach, a multistart approximate shortest path algorithm and an algorithm based on a tripartite network. In the first two algorithms we used simple iterative approach to generate five campaigns one after another by maintaining a global array of already used nodes. We generate a graph where each slab represents a node and construct an arc between two slabs if they satisfy the difference in width and thickness requirement. We construct an arc from wide to narrow slab with a cost of 1 unit and from narrow to wide slab with a cost of 10 units. From now on we will use slabs and nodes interchangeably to refer to the same thing.

5.2.1 Nearest Neighbor

This approach creates the campaign by selecting the best available slab at every point of the algorithm. We start with a slab of highest grade available which is wide enough and does not have a very high earliest start time. We achieve this using a weighted combination of increase in make span and width. We introduce a slight randomness in this process in the sense that if a slab improves this weighted combination then it is chosen as the candidate and if it does not then it is chosen with a probability depending on how it affects the solution quality. To choose slabs at each iteration of the algorithm we try to minimize a weighted cost which apart from considering the increase in make span of the campaign also takes into account the grade and profile restrictions. Since the approach looks only one step ahead it is short sighted in the sense that we might end up at a slab after which we cannot schedule any slab feasibly.

5.2.2 Multistarts or Shortest Path Approach

In this approach we add a source node and a sink node to our graph and the aim is to find an approximate shortest path from source to the sink. Initially all nodes of highest grade and sufficiently high width (this parameter can be fixed depending on the campaign to be built) are made reachable from the source node and the cost of reaching a node depends on its earliest start time and processing time, thus allowing the possibility of multiple starting points to generate the campaign. The cost of remaining arcs in the graph is determined dynamically as we go along in the algorithm.

We compute and update the distance of not yet visited nodes from the visited nodes at every step of the algorithm. The distance is computed as a weighted average of increase in make span and profile restrictions and is called the effective cost. Note that the path taken to reach a node influences the computation of effective cost. While updating and computing cost of unvisited nodes we decrease the effective cost

of reaching a node in proportion to the cumulative weight of the path taken to reach it. This promotes generation of longer paths and eventually fills up the campaign to the required weight. Note however that this makes some of the arc costs negative. At every node reachable from the source node we maintain the cumulative weight, time span and effective cost. All nodes with cumulative weight 30000 (the required minimum weight of the campaign) are connected to the sink node with 0 cost arcs.

5.2.3 Tripartite Approach

We developed and implemented an algorithm based on the tripartite network of [12] where the problem of assigning slabs to campaigns is formulated as a minimum cost flow problem. Though our algorithm differs from the approach of [12] in various aspects it shares the same underlying network. In [12] campaigns to be built are chosen one by one (a minimum cost flow problem is solved each time)from a set of possible candidates. In our approach we fix the number of campaigns we wish to generate and then build them by solving a single minimum cost flow problem. Also the problem of scheduling slabs within a zone to obtain campaign's schedule is not addresed in [12]. Algorithm 1 presents an outline of our approach.

Algorithm 1 A Tripartite Network Based Algorithm

Procedure Tripartite Algorithm

Construct the tripartite network T.
Find a minimum cost flow F in T.
for all slabs i **do**
  Assign slab i to a campaign C based on F.
end for
for all campaigns C **do**
  Schedule slabs assigned to C.
end for

Graph Generation

The first level of nodes in the network represent slabs (we make one node for each slab i). Each slab node i acts as a source providing $weight(i)$ units of flow. In the second layer, also called the zone layer, a fixed number of nodes called the zone nodes are generated for each campaign we wish to make. Each zone node represents a particular zone of the corresponding campaign. The third level contains one node for each campaign to be built. Finally we construct a sink node with demand equal to the sum of weights of all slabs and a by-pass node to carry flow from the unused slab nodes to the sink node. An arc connects a slab node to a zone node if it is permissible for the slab to be placed in the corresponding campaign in that zone. Each zone node has an outgoing arc to its corresponding campaign node and each campaign node has an outgoing arc to the sink node. All slab nodes are connected to the by-pass node. If a slab is used then we get a flow from slab node to the corresponding zone and campaign node and if it is not used then it passes thru the by-pass node. Figure 2 represents the tripartite network. Entries on nodes represent the demand/supply for that node. On the arcs entries inside the square brackets represent lower and upper capacity and the entry outside the bracket represents the cost of the arc.

We experimented with two different approaches for deciding on which campaigns to build for our problem data. These approaches help in defining the arc capacities and costs.

In the first approach we generate 5 campaigns in different time zones. Time stamps on campaigns help us to categorize slabs into campaigns and to set cost on arcs from slab nodes to zone nodes. Each campaign is divided into 15 width zones uniformly over the entire width range of slabs. The lower capacity on arcs from zone nodes to campaign nodes is set to a non zero value to ensure picking slabs from every width zone. A sufficiently high upper capacity on these arcs allows for various profiles ranging from ones that have lot of high width slabs to one that have high number of low width slabs. Capacity on arcs from campaign nodes to

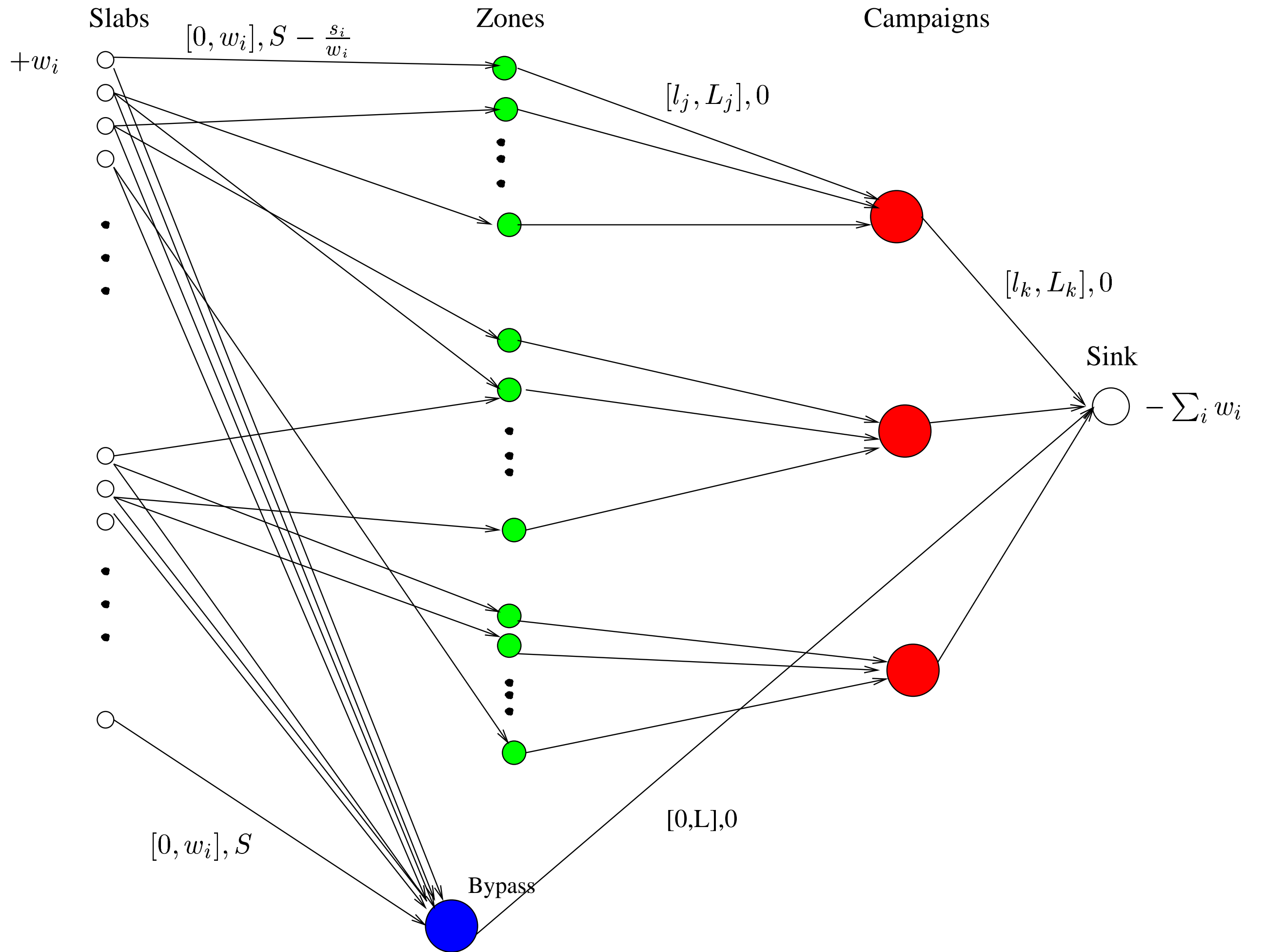


Figure 2: Minimum cost flow network for allocating slabs to campaigns

sink node is set to ensure filling the campaigns with minimum required weight. To achieve the grade ratio and profile of slabs in a campaign we do not assign higher grade slabs to the end zones. Each slab node i is connected to the by-pass node with a very high cost arc, say cost is S, and capacity $[0, weight(i)]$. The cost on the arcs from slab node i to a zone node j is

$$C - \frac{s_{ij}}{weight(i)} \text{ where}$$

s_{ij} is assigned a high value if $EST(i)$ is less than the time stamp on the campaign node corresponding to the zone node j otherwise it is assigned a low value. Thus if the slab is used then all $weight(i)$ units flow from a slab node through a campaign node, then this flow will be cheaper by s_{ij} units than if it passes thru the by-pass node. We need to be very careful about assigning upper and lower capacities to arcs as we might end up in no feasible flow if they are not properly chosen.

As a different approach we generate 5 campaigns with different starting width zones. Each campaign is assigned 8 width zones and each width zone covers a range of 100 width units. For example a campaign with starting width zone $[2500 - 2400]$ will have $[1800 - 1700]$ as its last width zone. Zone nodes that correspond to width ranges that can be assigned to fewer(more) campaigns are connected to the campaign nodes via higher(lower) capacity arcs. Capacity and cost issues on other arcs are addresed in the same way as is the previous approach. Note that each slab node can be assigned to atmost one zone node of a campaign depending on its width in both the approaches. We wish to promote (discourage) assignment of slabs with low(high) EST to earlier zones in the campaign. The cost on arcs, from slab nodes to zone nodes, is thus calculated by measuring the difference between zones that are suitable with time and width considerations.

Post Processing

The solution of the tripartite approach only assigns slabs to campaigns. It needs to be processed to create the final schedule. We assign a slab to the campaign that has more than half of the flow from the corresponding slab node assigned to one of its zone node. Zones of a campaign are scheduled one after another. To schedule slabs within a zone we try to maintain feasibility with respect to the maximum thickness difference and maximum width difference constraint between any two consecutive slabs with an objective of minimizing the time span. If within a zone no feasible slab is found then the slab that causes the least increase in the time span of the campaign is scheduled. Campaign is started with the earliest available slab in its first zone.

Note that to schedule the slabs within a zone any ATSP heuristic can be used. We used a simple nearest neighbor based heuristic and a patch heuristic based on the solution of the assignment problem. The motivation to use assignment problem based heuristic is that, as evident from the previous section, it performs better than other heuristics for random ATSP and from literature it is evident that it performs well for some steel industry related flow shop problems [26]. For the AP based patch heuristic a bipartite network B is constructed. For each slab i assigned to the zone two nodes $i.first$ and $i.second$, one in each partition, are constructed. Width and thickness of slabs govern the construction of arcs and start dates govern the arc costs. If $EST(i) < EST(j)$ and $|EST(i) - EST(j)|$ is less than a predefined value than cost on arc $(i.first, j.second)$ is defined to be $EST(i) - EST(j)$ else it is assigned to be $|EST(i) - EST(j)|$. This cost structure makes B asymmetric. Note that if both the arcs $(i.first, j.second)$ and $(j.first, i.second)$ are assigned the same cost then we will end up with too many two cycles in the solution to the matching problem. To ensure a perfect matching we add edges between $(i.first, i.second)$ for all i with a high cost. We then find a minimum weight perfect matching in B and patch the cycles greedily to obtain the final schedule.

The problem of scheduling slabs assigned to one zone can also be seen as the single machine scheduling problem where we wish to minimize the makespan. Every job here has a release date, processing time, a due date and feasibility constraint goverened by thickness and width of slabs.

5.3 Local Search

We tried to improve on the campaigns obtained from nearest neighbor and shortest path approaches by using Balas [5] algorithm for precedence constraint TSP. Empirical results obtained very high values (upto 200) for the constraint parameter $k(i)$. $k(i)$ is the parameter such that

$$\pi(i) < \pi(j) \forall i, j \in \sigma \text{ such that } i + k(i) \leq j$$

where σ is the initial configuration of campaign and π is the final configuration. We used width of slabs to define k values. Very high values of k reflects that we need to build better campaigns to be able to apply this algorithm.

We propose a local search heuristic similar to the approach of [1]. The basic idea is to guide the search for improving moves thru a graph called the improvement graph. A negative cost $s-t$ path in this graph corresponds to a set of moves which would result in an improvement, in terms of reducing infeasibilities, obtaining better profile and decreasing time span, of the overall schedule of all campaigns. To improve the schedule within a campaign we propose to apply 3-opt heuristic after every certain number of iterations of local search. Note that to maintain the grade restrictions 3-opt can be applied independently only in different grade zones of a campaign. This structure can very easily be embedded within a tabu search or other metaheuristic framework to obtain better results. The general local search structure is described in algorithm 2.

Algorithm 2 A Local Search Algorithm

```
Procedure Local Search
  Construct an initial solution S.
  Construct improvement graph G_S.
  while there is a negative length path P in G_S do
    Apply the moves induced by path P to S.
    Update S and G_S.
  end while
```

5.3.1 Neighborhood

In local search approaches it is necessary to consider large enough neighborhood so as to not get trapped in small local optimas but the larger the neighborhood the harder it becomes to search it. For the HSM problem we now present the structure of a large scale neighborhood and an efficient method to search it. The large scale neighborhood we propose can perform mulitple inserts, deletes and swaps between different campaigns and from the unused slab pool simultaneously.

Let us say n represents the number of slabs and m represents the number of campaigns to be formed. We denote a slab i from the unused pool as i_0 and the ith slab in the rth campaign as i_r. We consider the following three types of moves:

1. $Insert1[i_r, j_s]$: Slab represented by i_r is moved from campaign r to the $j-1$th position in campaign s.
2. $Insert2[(i_0, j_s]$: Slab i from the unused pool is inserted in campaign s at the $j-1$th position.
3. $Delete[(i_r, j_0]$: ith slab is deleted from campaign r and slab j_0 is such that in a feasible campaign it would be placed somewhere after the slab represesnted by i_r.

Whether a slab i would be placed before slab j or after it can be decided by comparing their width, grade and release times. Two moves say, $Insert1[i_r, j_s]$ and $Insert1[k_p, l_q]$, are said to be *independent* if in any

feasible schedule slab represented by j_s would be placed before slab k_p or slab l_q would be placed before slab represented by i_r. Similarly we define independence for other moves. Our neighborhood consists of all the schedules that can be obtained from the current one by applying a set of independent moves.

5.3.2 Improvement Graph

Improvement graphs were first used for large scale local search in [31]. Since then the concept has been used successfully for many scheduling problems. For a given solution S we create a directed acyclic graph $G_S(V, A)$. We construct two nodes $v(i_r)$ and $v'(i_r)$ for the ith slab in the rth campaign, a node $v(i_0)$ for every unscheduled slab i and a node $v(c_r)$ for $r = 1$ to m for every campaign r. Also the graph has two dummy nodes s and t. The arc set A consists of following arcs:

1. Arcs $(v'(i_r), v(j_s))$ exist for $r \neq s$ and represents that if slab i_r is re-assigned to campaign s it would be processed between slab $(j-1)_s$ and j_s. These arcs represent the move $Insert1[i_r, j_s]$.
2. Arcs $(v'(i_r), v(c_s))$ exist for all $r \neq s$ and represent that if slab i_r is removed from campaign r and is assigned to campaign s then it would be scheduled at the end of campaign s.
3. Arcs $(v(i_0), v(j_r))$ signify that slab i from the unused slab pool will be scheduled in campaign r at the $j-1$th position. These arcs represent the move $Insert2[i_0, j_r]$.
4. Arcs $(v'(i_r), v(j_0))$ represent that the slab represented by i_r is removed from the rth campaign and it goes to the unused slab pool. These arcs represent the move $Delete[i_r, j_0]$.
5. Arcs $(v(i_r), v'(i_r))$ signify that slab i_r will be part of an insertion move.
6. Arcs $(s, v(i_r))$ where $0 \leq r \leq m$ exist for all slabs(scheduled and unscheduled) and signify that slab represented by i_r will be part of an insertion move. There are $O(n)$ such arcs.
7. Arcs $(v(i_r), v(j_s))$ exist for $r \neq s$ if the suitable place for slab i_r in campaign s is right before the slab represented by j_s. These arcs facilitate moving from one campaign to another (ensurign acyclicity) in G_S in order to possibly start new insertion moves.
8. Arcs $(v(i_r), v((i+1)_r)))$ facilitate movement in G_S.
9. Arcs $(v(n_r^k), v(c_r))$ exist for all r where n_k is equal to the number of slabs assigned to campaign r in the current schedule. These arcs facilitate movement on graph G_S. There are $O(m)$ such arcs.
10. Arcs $(v(c_r), t)$ for all r signify the end of the $s-t$ path . There are $O(m)$ such arcs.
11. Arcs $(v(i_0), t)$ for all unscheduled slabs signify the end of the $s-t$ path.

It is an easy observation to make that the graph G_S is acyclic by construction. Arcs that represent actual movement are always added from a node i to a node which represents a slab that would appear after the slab represented by node i in any feasible campaign. Note that the improvement graph successfully includes all the simple inserts, deletes and swap moves in any schedule. For example, consider a swap move involving the ith slab in campaign r and jth slab in campaign s. Let us say without loss of generality that in any feasible schedule i_r will be placed before j_s. The arcs $(v(i_r), v(k_s))$, $(v(j_s), v(l_r))$ and the 0 cost arcs from $v(k_s)$ to $v(j_s)$ represent this move, where $k \leq j$ and $l > i$.

Let M be the set of independent moves and let P be the path in G_S corresponding to these moves. Note that the effect on solution quality of a move from set M depends on the moves that have been made before it i.e. cost of an arc (i, j) in G_S depends on the path taken to reach node i say P_i. Arcs 1-4 signify actual movement of slabs and are the only non-zero cost arcs. We now define the cost on remaining arcs.

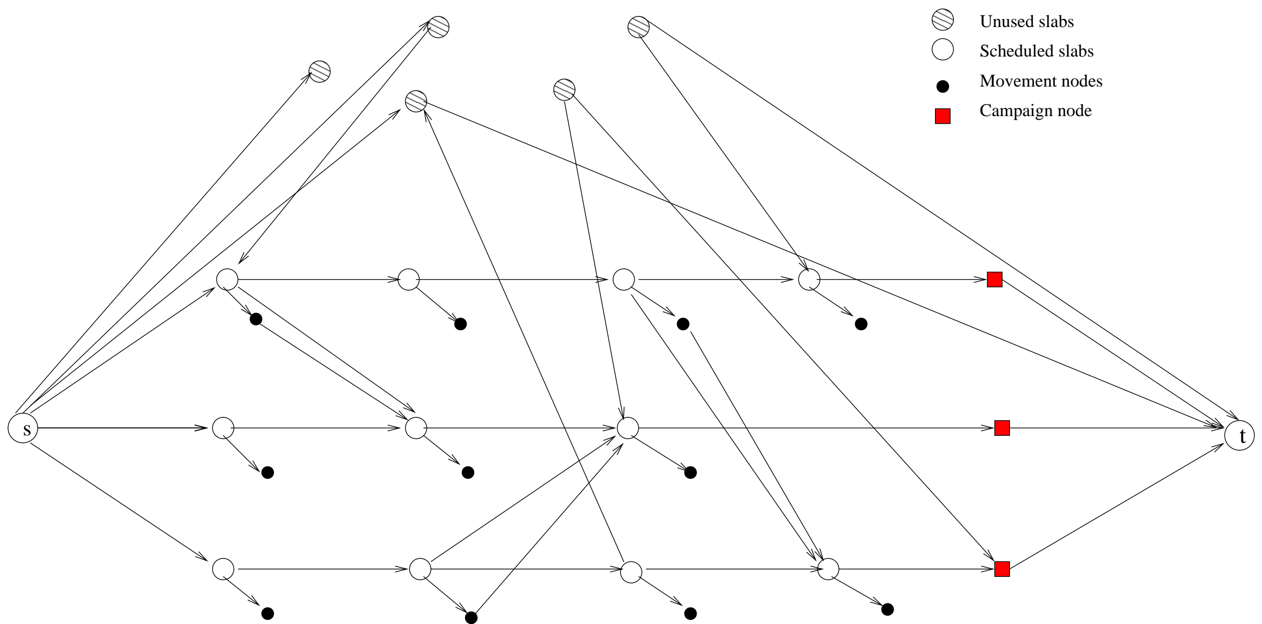


Figure 3: Structure of the improvement graph for the HSM problem

To compute the arc costs correctly and efficiently we maintain with each node v in G_S two labels - the change in makespan and the change in total weight of campaigns caused by the path taken to reach node v. We also associate with every node, that represents a scheduled slab, the cumulative weight of the campaign (uptil that point) in which it is scheduled and keep updating it to reflect the changes as a result of the path taken to reach that node. The cost on arcs can be defined, using these labels, as a weighted combination of change in make span and cumulative weight of the campaign. For example arc representing insertion of a slab in a campaign that has less than the minimum required cumulative weight or removal of a slab from a campaign that has a higher cumulative weight will have negative cost. Similarly an arc representing removal of a slab with very high wait time or causing infeasibility in a campaign should have negative cost. Insertion of a slab in a campaign which reduces its infeasibilities or yields a better profile should also have a negative cost.

5.3.3 Searching the Improvement graph

It can be proved [1] that searching the improvement graph exactly with the given path based cost structure for profitable moves is NP hard. An algorithm similar to the heuristic shortest path approach of [1] can be used to find a profitable $s-t$ path in G_S. In this approach nodes of the acyclic graph G_S are visited in the topological order and distance labels are updated at every step. Arc costs are determined dynamically as we move along in the algorithm. This is an $O(|V|.|A|)$ algorithm where $|V|$ represents number of nodes and $|A|$ number of arcs in the improvement graph.

5.4 Results

We generate instances with 5000 nodes and, with the problem specifications we consider, on an average number of arcs in the graph are 700,000. The implementation takes around 1 minute to generate the problem instance. We now present some of the results generated by various algorithms implemented by us. Whereas the shortest path algorithm was able to generate all 5 campaigns in all problem instances the nearest neighbor algorithm could generate all 5 campaigns in only 50% of the problem instances. In 30% of

the instances it could generate 4 campaigns and in the remaining instances it generated only 3 campaigns. Table 3 compares 3 full campaigns (i.e. campaigns that meet the minimum weight requirement) generated by these algorithms. All times are reported in seconds.

Nearest Neighbor				Shortest Path			
Time	Make Span	Cost	#Slabs used	Time	Make Span	Cost	#Slabs used
0.6	2362	2452	833	1.2	2336	2633	852
0.7	2468	2396	858	1.2	2428	2562	853
0.6	2630	2543	852	1.2	2396	2573	855

Table 3: Comparison between Nearest Neighbor and Shortest Path heuristics

A sample profile of width and thickness of campaigns generated by both the algorithms is shown in figure 4 and 5 respectively. The campaigns obtained by nearest neighbor and shortest path based heuristic didn't have any infeasibility in terms of difference in width and thickness between consecutive slabs. However the width profile of the campaigns generated by these algorithms are not very nice. Since the cost structure used in both the algorithms is same we get almost same profile. The main difference in two approaches is that in shortest path heuristic we can start from different slabs and continue along some path even though many other paths might fail to fill the campaign.

We now report on the results of various versions of the tripatrite approach. The approach of generating the tripartite network based on time stamps on campaigns could not generate the campaigns. However if we remove the grade restriction from the problem then this approach can generate campaigns. This is because of the fact that many slabs didn't get assigned to any zone node for e.g. slabs with width less than 1800(1500) that have high(medium) grade. The second approach of constructing the tripartite graph can generate campaigns in most of the instances. We now report results for this approach.

We scheduled the slabs within a zone using two different algorithms - Nearest neighbor and Assignment problem heuristic. The performance of both the algorithm is approximately the same as far as time span of the campaigns is considered. Though assignment problem solution had fewer number of infeasibilities (in terms of difference in width and thickness between consective slabs) as compared to the nearest neighbor solution. The fact that assignment problem based heuristic performs far better than the nearest neighbor heuritic for ATSP and not for our problem is due to the reason that we do not have a complete graph in our situation. An analysis of the result obtained from the assignment problem shows that it could not generate bigger cycles and had to resort to greedy algorithm many times to patch the solution to form campaign. The time taken to process a campaign built by tripartite approach is slightly higher as compared to other two approaches however it yields a better profile for the campaign. One typical profile of width and thickness of the campaigns generated by tripartite approach is shown in figure 4 and 5.

The algorithm took about 0.3 seconds to generate the tripartite graph and 3-4 seconds to find the solution of the minimum cost flow problem. In table 4 we report the times taken(in seconds) by nearest neighbor and assignement problem approaches for scheduling the campaigns. Many a times the result of the tripartite approach was not able to fill the campaign upto 30000 tons but it always remained very close to this figure.

Nearest Neighbor				Assignment Problem			
Time	Make Span	Weight	#Infeasibilities	Time	Make Span	Weight	#Infeasibilities
0.4	2569	29200	33	1.2	2598	29200	21
0.2	2727	30026	31	1.0	2686	30026	26
0.3	2478	29799	35	1.0	2484	29799	24

Table 4: Comparison of campaigns generated by tripartite approach using nearest neighbor and assignemnt problem solution for scheduling

The rounding scheme adopted by us to assign slabs to campaigns performs well and the gap, between the cost obtained by the solution of the min cost flow problem and the solution obatined by the rounding scheme, was always less then $10^{-3}\%$ in our experiments. In the tripartite approach while scheduling the nodes within a zone to form a campaign we often landed in situations where no slab could be feasibly scheduled from among the slabs assigned to the campaign. Most of the infeasibilities arise from the thickness constraints rather than the width between consecutive slabs. On an average nearest neighbor based schedule reported 30 infeasibilities and assignment problem based schedule reportes 23 infeasibilities. We tried to reduce these infeasibilities by changing the scheduling process. In the nearest neighbor approach whenever we could not schedule any node in the zone feasibily we also considered the slabs assigned to next zone. This reduced infeasibilities from 30 to 12 on an average in a campaign. However this distorted the profile of the campaign as shown in figure **??**. The guarantee that no slab of higher grade will be scheduled towards the end of the campaigns is also lost.

An analysis of the unused nodes shows that mostly they are the ones that have width lower than 1300 and grade 2, the ones with width less than 1100 and grade 1 and the ones that have width more than 2200 and very high EST.

5.4.1 Concerns Regarding Data

Some of the slabs badly conflict with the objectives and the constraints of the problem. Following are some of the cases:

1. A slab with low width and high grade is diffciult to schedule in any campaign since we cannot allocate any high grade slab towards the end of the campaign and because the width is low hence it cannot be allocated at the start of the campaign.

2. A very low width slab with low EST cannot be scheduled at the start of any campaign whereas it would be good to schedule it at the start considering the time span of the campaign.

3. The time structure on slabs is also not very good. All of them are generated with 1440 minutes and have a maximum LST of 1920(1440+480). In all of the algorithms we took atleast 2200 minutes to schedule a campaign. If we wish to schedule campaigns one after another to generate the overall schedule of the steel mill then basically the EST of slabs will not make a difference after the first campaign (because we will not have to wait before processing any slab) and we will not be able to meet LST for slabs after the first campaign.

References

[1] R. Agarwal, O. Ergun, J. B. Orlin and C. N. Potts, Solving parallel machine scheduling problems with large scale neighborhoods search, *In preparation*, 2004.

[2] K. Altinkemer and B. Gavish, Parallel savings based heuristic for the delivery problem, *Operations Research*, **39**, 456-469, 1991.

[3] D. Appelgate, R.E. Bixby, V.Chvatal and W. Cook, On the solution of travelling salesman problems, *Documenta Mathematica*, Extre Volume **ICM III**, 645-656, 1998. The 12/15/1999 release of the *Concorde* code is currently available from http://www.math/princeton.edu/tsp/concorde.html

[4] E. Balas and C. H. Martin, Combinatorial optimization in steel rolling, *DIMACS/RUTCOR workshop proceedings*, 1991.

[5] E. Balas and N. Simonettii, Linear time dynamic programming algorithms for new classes of restricted TSP's: A computational study, 2000.

[6] J. L. Bentley, Multidimensional binary search trees used for associative search, *Comm. ACM*, **18**, 509-517, 1975.

[7] R. Bent and P. V. Hentenryck, A two stage hybrid local search for the vehicle routing problem with time windows, *Technical Report*, **CS-01-06**, Department of Computer Science, Brown University, September 2001.

[8] J. B. Bramel and D. Simchi-Levi, A location based heuristic for general routing problems, *Operations Research*, **43**, 649-660, 1995.

[9] G. Clarke and J. R. Wright, Scheduling of vehicles from a central depot to anumber of delivery points. *Operations Research*, **12**, 568-581, 1964.

[10] N. Christofides, Worst case analysis of a new heuristic for the travelling salesman problem, Report No. 388, GSIA, Carnegie Mellon University, Pittsburgh, PA, 1976.

[11] N. Christofides, A. Mingozzi and P. Toth, The vehicle routing problem. In: N. Christofides, A. Mingozzi, P. Toth and C. Sandi(eds) *Combinatorial Optimization*, 315-338, 1979.

[12] P. Cowling and W. Rezig, Integration of continuous caster and hot strip mill planning for steel production, *Journal of Scheduling*, **3**, 185-208, 2000.

[13] J. F. Cordeau, G. Laporte and A. Mercier, A unified tabu search heuristic for vehicle routing problems with time windows, *Journal of Operations Research Society*, **52**, 928-936, 2001.

[14] G. A. Croes, A method for solving travelling salesman problems, *Operations Research*, **6**, 791-812, 1958.

[15] M. L. Fisher and R. Jaikumar, A generalized assignment heuristic for vehicle routing, *Networks*, **11**, 109-124, 1986.

[16] A. Frieze, G. Galbiati and F. Maffoli, On the worst-case performance of some algorithms for the asymmetric travelling salesman problem, *Networks*, **12**, 23-39, 1982.

[17] M. L. Fredman, D. S. Johnson, L. A. McGeoch, G. Ostheimer, Data structures for travelling salesman, *Journal of Algorithms*, **18**, 432-479, 1995.

[18] B. E. Gillett and L. R. Miller, A heuristic algorithm for the vehicle dispatch problem, *Operations Research*, **22**, 340-349, 1974.

[19] F. Glover, G. Gutin, A. Yeo and A. Zverovich, Construction heuristics and domination analysis for the asymmetric TSP, *European Journal of Opertaions Research*, **129**, 555-568, 2001.

[20] M. Held and R. M. Karp, The travelling salesman problem and minimum spanning trees, *Operations Research*, **18**, 1138-1162, 1970.

[21] M. Held and R. M. Karp, The travelling salesman problem and minimum spanning trees: Part II, *Math Programming*, **1**, 6-25, 1971.

[22] J. Homberger and H. Gehring, Two evolutionary metaheuristics for the vehicle routing problem with time windows, *INFOR*, **37**, 297-318, 1999.

[23] Computer solutions of the travelling salesman problem, *Bell System Tech. Journal*, **44**, 2245-2269, 1965.

[24] S. Lin and B. W. Kernighan, An effective heuristic algorithm for the travelling salesman problem, *Operations Reserach*, **21**, 498-516, 1973.

[25] D. S. Johnson, J. L. Bentley, L. A. McGeoch and E. E. Rothberg, Near optimal solutions to very large travelling salesman problems, *in preparation.*

[26] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang and A. Zverovitch, Experimental analysis of heuristics for the ATSP, *The Travelling Salesman Problem and its Variations*, Gutin and Punnen(eds), Kluwer Academic Publishers, 445-487, 2002.

[27] P. C. Kanellakis and C. Papadimitriou, Local search for the asymmetric travelling salesman problem, *Operations Research*, **28(5)**, 1086-1099, 1980.

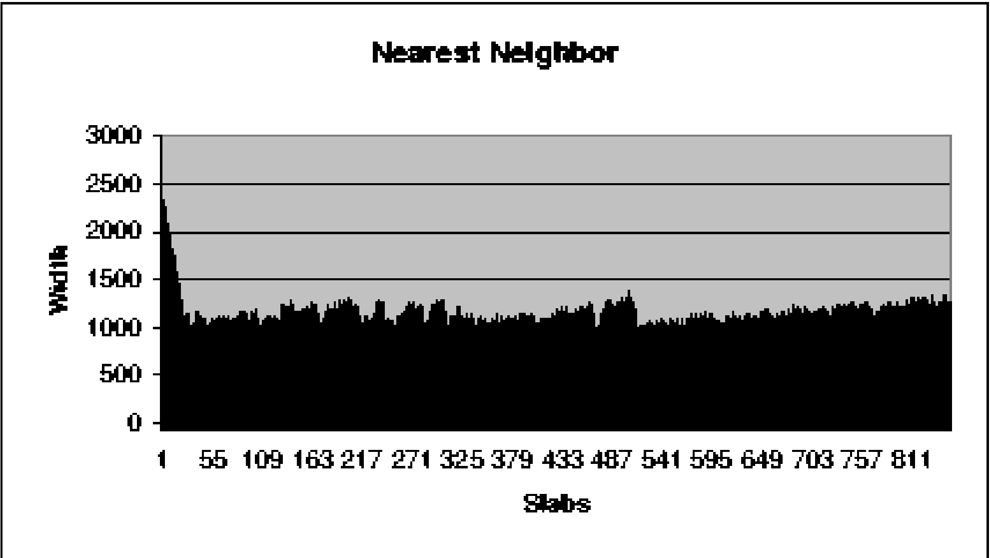
[28] R. M. Karp and J. M. Steele, Probabilistic analysis of heuristics, *The travelling salesman problem: A guided tour of combinatorial optimization*, 1985.

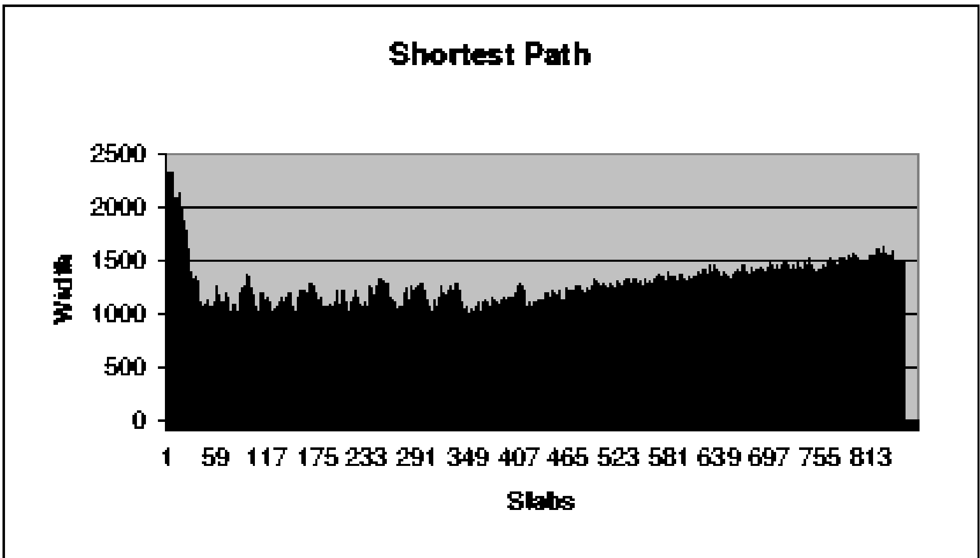
[29] Y. Rochat and E. D. Taillard, Probabilistic diversifictaion and intensification in local search for vehicle routing, *Journal of Heuristics*, **1**, 147-167, 1995.

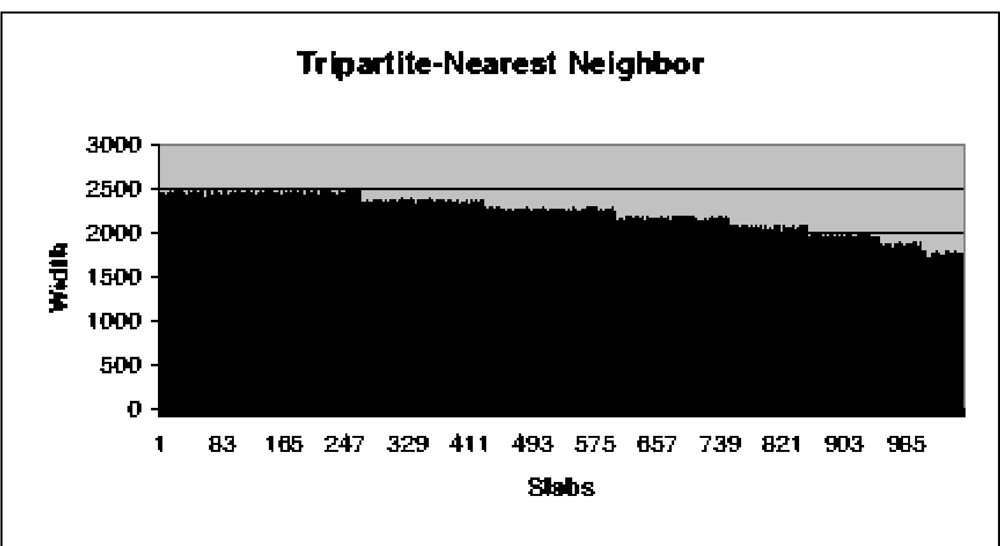
[30] M. M. Solomon, Algorithms for the vehicle routing and scheduling problems with time window constraints, *Operations Research*, **35(2)**, 1987.

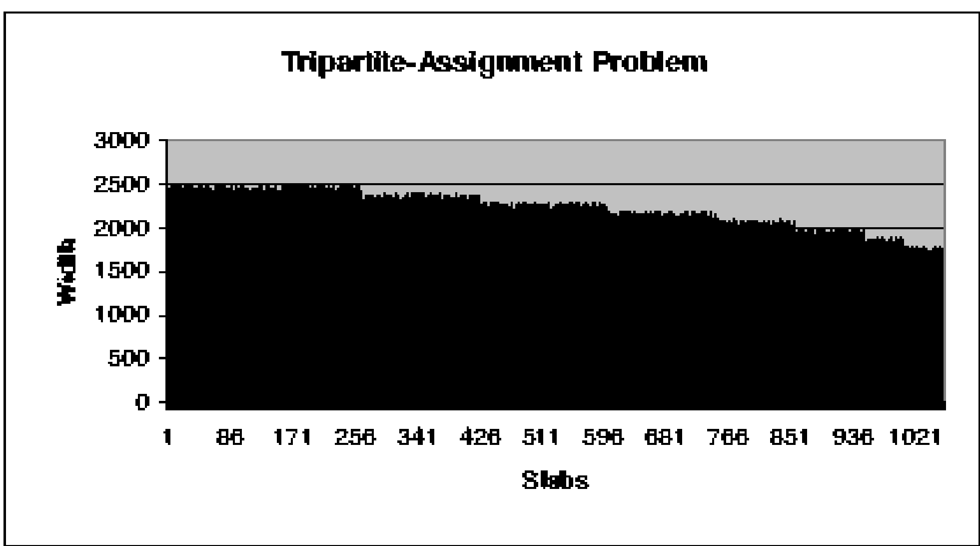
[31] P. M. Thompson and J. B. Orlin, The theory of cyclic transfers, *Operations Reserach Center Working Paper, MIT OR200-89*, August 1989.

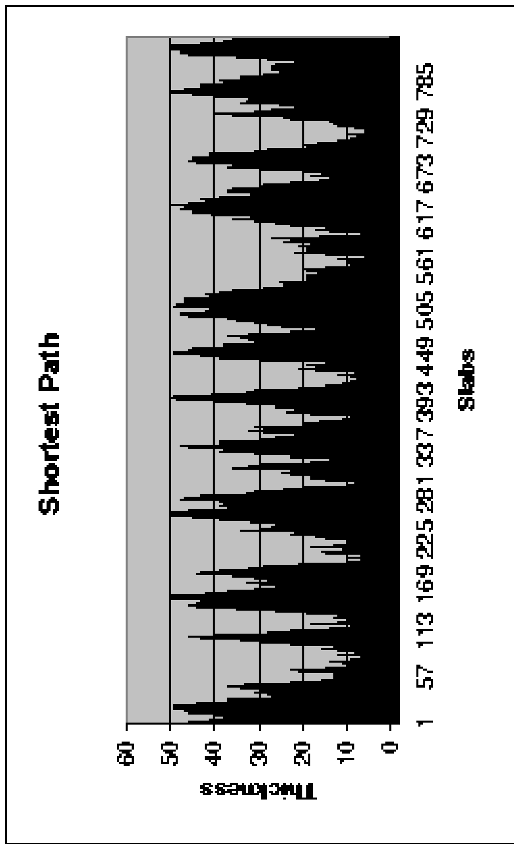
[32] H. Yasuda, H. Tokuyama and K. Tarui, Two stage algorithm for production scheduling of hot strip mill, *Operational Research*, **J. B. Barns(editor)**, 1984.

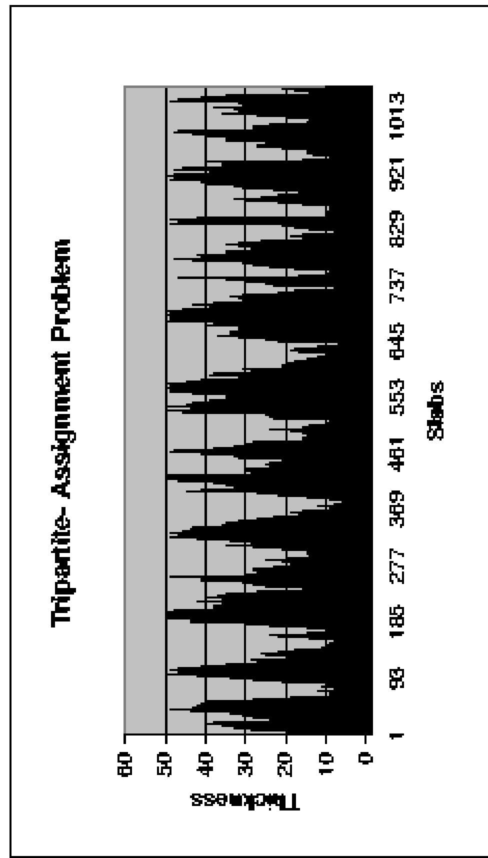

Nearest Neighbor
Width
3000
2500
2000
1500
1000
500
0
1 55 109 163 217 271 325 379 433 487 541 595 649 703 757 811
Slabs

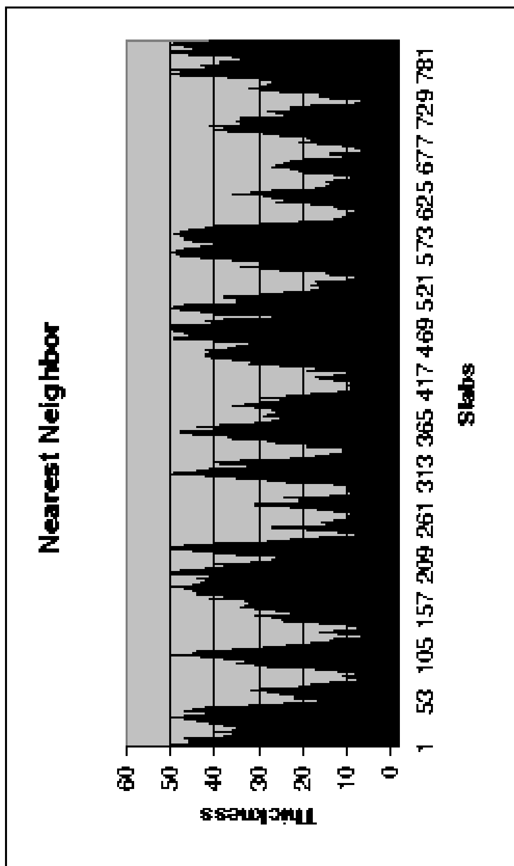

Shortest Path
Width
2500
2000
1500
1000
500
0
1 59 117 175 233 291 349 407 465 523 581 639 697 755 813
Slabs


Tripartite-Nearest Neighbor
Width
3000
2500
2000
1500
1000
500
0
1 83 165 247 329 411 493 575 657 739 821 903 985
Slabs


Tripartite-Assignment Problem
Width
3000
2500
2000
1500
1000
500
0
1 86 171 256 341 426 511 596 681 766 851 936 1021
Slabs







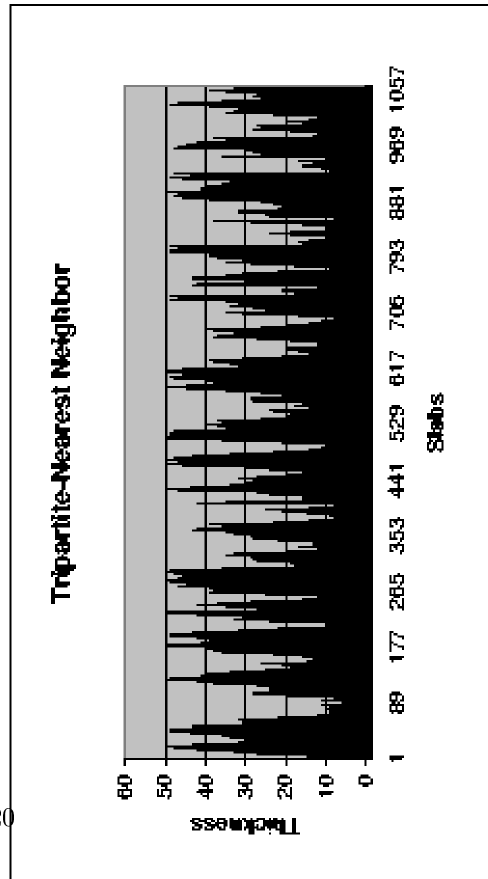


Figure 5: Thickness profile of campaigns generated by different algorithms

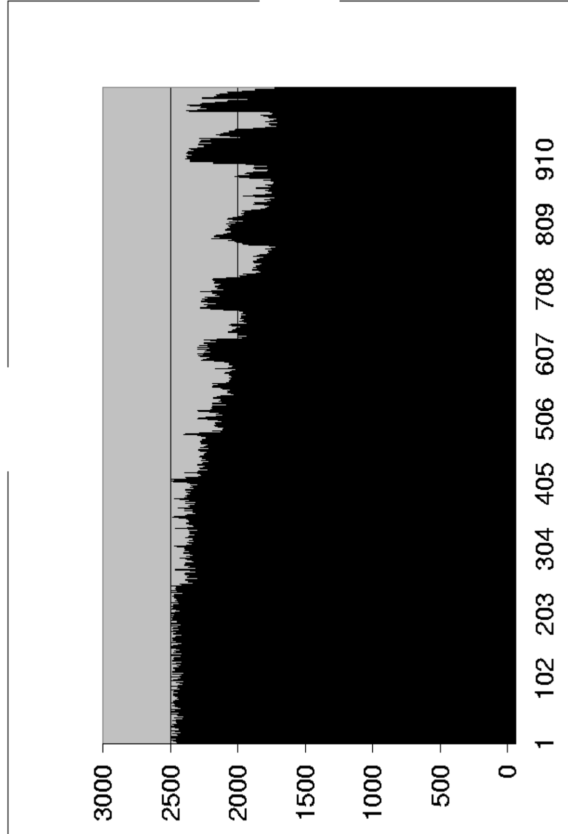


Figure 6: Width profile of campaign generated by reduced infaesibilities version of tripartite approach