TCBB Scheme: Applications to Single Machine Job Sequencing Problems

Sakib A. Mondal

Infosys Technologies Limited
27, Bannerghatta Road, JP Nagar 3rd Phase
Bangalore 560 076, INDIA
AbdulSakib@inf.com

Anup K. Sen

School of Management
New Jersey Institute of Technology
University Heights, Newark, NJ 07102, USA
sen@njit.edu

Abstract

Transpose-and-Cache Branch-and-Bound (TCBB) has shown promise in solving large single machine quadratic penalty problems. There exist other classes of single machine job sequencing problems which are of more practical importance and which are also of considerable interest in the area of AI search. In the weighted earliness tardiness problem (WET), the best known heuristic estimate is not consistent; this is contrary to the general belief about relaxation-based heuristic. In the quadratic penalty problem involving setup times (SQP) of jobs, the evaluation function is non-order-preserving In this paper, we present the TCBB scheme to solve these problems as well. Experiments indicate that (i) for the WET problem, the TCBB scheme is highly effective in solving large problem instances and (ii) for the SQP problem, it can solve larger instances than algorithm GREC in a given available memory.

Introduction

Best-first search algorithms like A* require substantial memory to store the generated nodes. On the other hand, depth-first search algorithm uses memory linear in the depth of the search and has very low overhead but runs slower than best-first in search graphs due to generation of duplicate nodes. The trend now is to look for variants of depth-first search which would run in reasonable time to solve problems of larger sizes.

Kaindl et al. (1995) proposed the *Transpose-and-Cache Branch-and-bound* (TCBB) scheme to solve large instances of single machine quadratic penalty job sequencing (QP) problem. The TCBB scheme is a variant of the depth-first branch-and-bound (DFBB) scheme. The DFBB uses a tree search space and it does not utilize the large amount of memory available. The TCBB overcomes this limitation by storing the nodes in the available memory and thus tries to avoid generation of duplicate nodes. This speeds up DFBB considerably. However when memory becomes full, TCBB continues as in DFBB and does not employ any node replacement strategies.

In the weighted earliness-tardiness (WET) problem, jobs with due dates are to be sequenced on a machine such that sum of earliness and tardiness penalty is minimized. A job completing earlier than the due date incurs an earliness penalty (inventory carrying cost) whereas a job completing later incurs a tardiness penalty (imposed by the customer). Currently known best approach for solving this problem is dynamic programming (Ventura and Weng 1995) which runs quickly out of memory. When the penalty coefficients are unity, a branch-and-bound tree search formulation (Hoogeven, Oosterhout, and Van De Velde 1994) has been suggested. Contrary to the general belief (Pearl 1984), the proposed Lagrangian-relaxation based heuristic for the problem is *not consistent*. When the heuristic is not consistent, A* graph search performs poorly since a node may be expanded more than once.

The other problem we consider in this paper is the single machine quadratic penalty problem (SQP) which includes setup times of jobs (Sen and Bagchi 1996). The presence of setup times makes the evaluation function *non-order preserving* (Pearl, 1984) and algorithm A* using graph search space is unsuitable for the problem since no path to a node can be discarded. Algorithm GREC has been suggested by Sen and Bagchi (1996) for finding optimal solutions using graph search space. GREC could solve 20-job problems using a hash queue of 200k nodes and runs faster than tree search, but like A*, it runs out of memory for solving large problems.

In this paper, we employ the TCBB scheme to solve optimally larger instances of the WET and the SQP problems. Instead of using g-values and f-values of nodes (Kaindl et al. 1995), the scheme is rewritten using b-values of nodes to naturally back up values obtained during the search process (Sen and Bagchi 1989). The use of backed-up values can be found in a number of algorithms (Kaindl and Kainz 1997). The use of b-values not only retains the heuristic improvement feature of the algorithm but also makes the approach applicable for the SQP problem. In addition, the scheme is enhanced with a node replacement strategy which would replace the less promising nodes (Reinefeld and Marsland 1994) when memory becomes full. Experimental results encourage the use of the TCBB scheme for the problems considered in the paper.

First, we describe the TCBB scheme and present the code for the algorithm. We then evaluate its performance in the QP problem domain. Next, we describe the WET and the SQP problems respectively, and present our experimental findings. Concluding remarks are given at the end.

The TCBB Scheme

The TCBB scheme is based on the DFBB algorithm. The scheme starts with an upper bound on the cost of the optimal solution and examines all the paths from the start to the goal node in a depth-first manner pruning paths of costs higher than the current upper bound. DFBB never attempts to store the nodes generated except that the nodes on the current path are stored in an implicit stack. On the other hand, TCBB attempts to store the expanded nodes along with their h-values and currently known best g-values if memory is available. Hence when an already stored node is encountered while searching along a different path, TCBB can use the stored g-value and decide whether the node is to be searched again. Thus like A*, it can discard paths to a node if the paths are of higher cost. This approach introduces the graph search feature in the algorithm and the performance of the algorithm becomes comparable to that of A*. The scheme utilizes a transposition table for recognizing transpositions and for caching the best values acquired dynamically (Reinefeld and Marsland 1994).

Unlike A* when memory becomes full, TCBB can still continue as in DFBB and can output optimal solution. Though TCBB cannot store new nodes in such situations, it can still continue to take advantage of already stored nodes. This feature of the scheme may make it more attractive than A*. In the QP problem domain, TCBB was shown to solve 60-job problems within reasonable time.

However, the TCBB scheme can be improved further. For very large problem instances when memory becomes full, use of effective memory management schemes may speed up the search process. Secondly, using g-values of nodes, a graph search algorithm is likely to maintain only the currently known least cost path to a node which in non-order-preserving cases, may lead to nonoptimal solutions. Instead, the scheme can be rewritten using b-values of nodes and arc costs below nodes. The b-value b(n) of a node n stores the cost of the currently known best path below node n. The use of b-values and arc costs not only helps in non-order-preserving cases but also helps in naturally incorporating dynamic heuristic improvements when more accurate estimate is obtained through the search experience (Kaindl and Kainz 1997, pp. 287). The revised TCBB scheme is presented below.

Enhanced TCBB scheme

The enhanced version of the scheme is given in Figure 1. For ready reference, the new scheme is called as *Enhanced TCBB (ETCBB)*. We explain below the working of the scheme in terms of b-values of nodes and arc costs.

The b-value b(n) of a node n store the cost of the currently known best path below node n. When node n is generated for the first time b(n) is set to h(n); b(n) increases whenever the cost of the currently known best path below

```
function ETCBBB(n: node; bound: integer): integer;
var  newbound: integer;
     minnode: node;
begin
   if n is a goal node then
   begin    label n SOLVED;
         return 0;
   end;

   newbound := ∝;
   for every successor $n_i$ of n do
   begin
      if $n_i$ is not a stored node then
         $b(n_i) := h(n_i)$;
      if $c(n,n_i) + b(n_i) < bound$ then
         if $n_i$ is NOT SOLVED then
         begin
            $b(n_i)$ := ETCBB($n_i$, bound - $c(n,n_i)$);
            if $n_i$ is not a stored node then
               SAVE($n_i$); /* Replace non promising node if
                                    memory full */
         end;
      if $c(n,n_i) + b(n_i) < newbound$ then
      begin    newbound := $c(n,n_i) + b(n_i)$;
               minnode := $n_i$;
      end
      else if $c(n,n_i) + b(n_i) = newbound$ and $n_i$ is SOLVED
           then     minnode := $n_i$;   /*preference to
                                              SOLVED successor */
      bound := min(bound, newbound);
   end;
   if minnode is SOLVED then label n SOLVED;
   return newbound;
end;
```

Figure 1: Algorithm ETCBB

node n (due to exploring the graph below node n) exceeds current value of b(n). As a result, if heuristic is admissible, h*(n) >= b(n)>= h(n) and hence use of b-value value helps avoiding repetitive search below n.

The algorithm uses the recursive function ETCBB to search the subgraph below the start node s. The current upper bound (we call it *bound*) on the cost of the subgraph below a node is passed as a parameter to the function. Let $c(n,n_i)$ denote the cost of the arc (n,n_i). If for a successor n_i of node n, $c(n,n_i) + b(n_i) <$ bound, ETCBB would search the subgraph below node n_i; the value (bound - $c(n,n_i)$) is passed as the current upper bound of the subgraph below the node n_i.

The local variable *newbound* in the function ETCBB stores the minimum of $c(n,n_i) + b(n_i)$, computed amongst all successors of node n. The value of newbound updates the value of b(n) on return from the function. A newbound lower than the current upper bound indicates that a better solution path has been found while a higher newbound indicates that all the paths below node n have been pruned.

During execution, newbound also resets the value of bound if it acquires a value lower than that of bound.

When memory becomes full, the simplest alternative is to continue as in DFBB. This strategy was followed in TCBB by Kaindl et al. For large problems, this scheme would run slow since no new nodes get stored and no advantage can be taken out of the stored values of new nodes. A better idea would be to replace the non-promising stored nodes with newly generated promising ones. This decision is critical. Nodes selected for replacement should not be frequently regenerated and reexpanded. In addition, the procedure for selecting nodes for replacement should not significantly increase the overhead of the algorithm. We made special experiments to evaluate a number of node replacement strategies. We found out that TCBB would perform better if, instead of continuing as in DFBB, attempts to replace less recently used nodes when memory becomes full.

ETCBB employs a SOLVE-labelling procedure (Pearl, 1984) if the evaluation function is order preserving. ETCBB labels a node SOLVED if the least cost path below the node has been found. When a SOLVED node is encountered along a different path, ETCBB can use its b-value to update the current bound or may decide to prune the current path. On the other hand, if a non SOLVED node is encountered, ETCBB can use its b-value to prune the current path or may peep below the node to search deeper into the subgraph below the node. We have experimented with or without the SOLVE-labelling procedure. When memory is large, there is hardly any effect of the procedure. But with a lower node limit of 2k nodes for the quadratic penalty (QP)[1] problem (Kaindl et al. 1995), the SOLVE-labelling procedure reduces number of nodes by 69% for 60-job problems, and by 72% for 62-job problems. This result is interesting. However, when the evaluation function is non-order preserving, the SOLVE-labelling procedure may not be in general applicable.

The performance of the ETCBB algorithm was found to be as good as, if not better, than the TCBB scheme for the quadratic penalty (QP) problem. Using reverse[2] search and the same consistent (Pearl 1984) heuristic estimate function, TCBB and ETCBB were run on DEC Alphastation 250 4/266 to solve problems of different sizes. For each problem size, both the algorithms solved the same 100 randomly generated problem instances. Table 1 presents a part of our experimental results when the node limit was fixed at 16k nodes (Kaindl et al 1995). Our experiments indicated the following:

Node limit = 16k nodes				
Job	TCBB		ETCBB	
	Node Gen.	Time (secs)	Node Gen	Time (secs)
56	139483	12.79	66543	6.94
58	181350	17.72	83723	9.35
60	243602	25.30	108558	13.06
62			131703	16.89

Table 1: TCBB and ETCBB for the QP problem

- ETCBB ran faster than the TCBB scheme generating less number of nodes. For 60-job problems, while the node reduction factor was 2.2, the speedup factor was nearly 2 due the overhead of the algorithm. TCBB could not be run to completion for some instances of 62-job problems since it took long time to execute.
- The node generated per second by both the versions are comparable For 60-job problems, TCBB generated 9626 nodes per second whereas ETCBB generated 8312 nodes per second. Since ETCBB has a higher overhead due to its node replacement strategy, it generated less number of nodes per second than that by TCBB.
- With a node limit of 2k nodes, ETCBB took 77 seconds to solve 62-job problems generating 552790 nodes. As expected, the node generation per second dropped to 7179 as more nodes were replaced due to lower availability of memory. TCBB took long time to complete even for 50-job problem instances.

Thus ETCBB appears to be a refined version of the TCBB scheme. We now investigate its applicability to two important classes of single machine job sequencing problems. We first present the weighted earliness-tardiness problem.

Weighted Earliness-Tardiness (WET) problem

In the weighted earliness-tardiness problem, jobs have due dates. A job completing earlier than the due date incurs an earliness penalty whereas a job completing later incurs a tardiness penalty. The objective is to schedule the jobs in such a way that the weighted sum of earliness and tardiness penalties is minimized. For the class of problem considered in this paper, the earliness and tardiness penalties are measured as the absolute deviation of job completion times around the common due date. The weights, that is, the earliness and tardiness penalty coefficients, depend on whether a job is early or tardy but do not depend on jobs. The problem can be formulated as follows: Given the processing times p_j of N jobs, the common due date d, and

[1] In the QP problem, a set of jobs J_i with processing times p_i, $1 \leq i \leq N$, are submitted to a machine at time t = 0. The jobs are to be processed on the machines one at a time. Let the processing of job J_i be completed at time C_i .The penalty functions are $\Psi_i(C_i)=\beta_i C_i^2$, $1 \leq i \leq N$ where β_i is the given positive penalty coefficient for job $J_{i.}$ The jobs must be sequenced in such a way that the sum of the penalties for all jobs are minimized.

[2] Kaindl et al (1995) had shown that due to asymmetric distribution of arc costs, searching in the reverse direction from the goal to the start yields much better results in this QP problem.

the earliness and tardiness penalty coefficients α and β respectively, the objective is to minimize $F = \Sigma_j(\alpha E_j + \beta T_j)$ where $E_j = \max(0,d-C_j)$ and $T_j = \max(0,C_j-d)$ are the earliness and tardiness of job J_j, and C_j denote the completion time of job J_j in a schedule.

The value of the common due date plays an important role in the complexity of the WET problem. When the due date is equal to or larger than a critical value d*, the WET problem becomes unrestricted; otherwise the problem is called restricted. The unrestricted WET problem is polynomially solvable whereas the restricted version is known to be NP-complete (Hall, Kubiak, and Sethi 1991). Other characteristics of the optimal schedule can also be found in (Hall, Kubiak, and Sethi 1991). Dynamic programming (DP) and Depth-first Branch-and-bound have been suggested so far for solving the restricted WET problem. These approaches can hardly be used in practice for solving large problem instances.

Hoogeveen, Oosterhout and Van De Velde (1994) suggested a *Lagrangian-relaxation* based heuristic estimate for the problem which is the best known heuristic estimate available in the literature. Interestingly, this admissible heuristic estimate is not consistent. It relaxes a constraint $W \leq d$ where W is the sum of the processing times of jobs completed before the due date. The heuristic is calculated as $\max_\lambda L(\lambda) = \max_\lambda[\min \{\Sigma_j (\alpha E_j + \beta T_j) + \lambda(W - d)\}]$ where $\lambda \geq 0$. $L(\lambda)$ can be found by Emmon's matching algorithm (Emmon 1987) with positional weights of early jobs increased by λ. Value of $L(\lambda)$ is maximized for that value of λ for which (W-d) changes sign from negative to positive. For a 5-job problem, let $p_1 = 12$, $p_2 = 26$, $p_3 = 32$, $p_4 = 53$ and $p_5 = 56$. Let $\alpha = \beta = 1$ and $d = 122$. Let for the start node s, h(s) = 161. Let n be an immediate successor of s where job J_5 is scheduled to complete at time = 179. Then c(s,n) = 57 and h(m) = 82. Thus the heuristic estimate is not consistent. If heuristic estimates are not consistent, algorithm A* expands a node more than once and performs poorly. Mero (1984) suggested a heuristic modification procedure to improve the performance of A* in such cases. However, this introduces more number of equal f-valued nodes and as a result, tie-resolution becomes an important issue.

We applied algorithm ETCBB to solve the restricted WET problem and achieved significant results. ETCBB could readily solve 2000-job problems using just 16k nodes of memory. Before we present our experimental results, we present salient features of our search formulation.

Search space

A node is represented in the search space as a tuple (U,I) where U represents the number of jobs yet to be scheduled and I is the time when the partial schedule consisting of U unscheduled jobs can start. Thus U can be stored as an index to the above ordered job sequence.

The start node is represented as (N,_) and its successor nodes as (N,0), (N,1), .., (N,d) since the optimal schedule can start at any time between 0 and d. Each of these successor nodes represent a subproblem to be solved. Any subproblem can be decomposed to at most two subproblems since every job may be scheduled either to the left or to the right of the due date. Two different subproblems when decomposed can lead to the same subproblem, thus resulting in a graph search space. For example, the nodes (3,0) and (3,1) may have the same common successor (2,2).

An arc in the search space represents the scheduling of a job and its cost may be taken as the incremental cost of scheduling the job. Therefore, the cost of the two successor arcs below a node would be α times the earliness of the job if the job is scheduled to complete before the due date, and would be β times the tardiness of the job if the job is scheduled to complete after the due date. In this representation, there can be atmost $d+p_N$ nodes at any level. As a result, the total number of nodes in the graph will be $O(N(d+p_N))$.

Successors are ordered in ETCBB following a heuristic rule. At a node n, a job would be scheduled before due date or after due date. Let n.left and n.right denote the unscheduled interval before and after due date at a node n. If $(\alpha + \lambda)$n.left > n.right, the first successor is generated by scheduling before due date else after due date. Heuristic estimates are computed following the Lagrangian-relaxation based heuristic suggested by Hoogeveen, Oosterhout and Van De Velde described above.

Experimental results

In our experiments, we fixed the memory as 16k nodes, and tested ETCBB, DP, DFBB and A* on DEC Alphastation 250 4/266 for problems with different due

Node limit = 16k nodes					
Job size	α: β	d=0.6d*		d=0.9d*	
		Node Gen.	Time (secs)	Node Gen	Time (secs)
100	5:1	846	0.07	946	0.09
	1:5	682	0.06	740	0.07
500	5:1	1570	0.95	1943	1.22
	1:5	1677	1.04	2197	1.35
1000	5:1	2764	3.47	3319	4.38
	1:5	3301	4.25	4149	5.39
1500	5:1	3082	6.75	5180	10.09
	1:5	4591	9.02	6662	12.87
2000	5:1	5540	14.07	6950	18.22
	1:5	6815	17.38	8663	22.75

Table 2: ETCBB for the WET problem

dates and with different ratios of α and β. Due date was fixed at d=kd*, 0 < k < 1 to solve restricted instances only. The problem criticality increases as d approaches d*, and for a fixed k, it also increases as the ratio of α and β decreases (Bagchi, Sullivan, and Chang 1987). Table 2 reports our findings for ETCBB and Table 3 for other algorithms. Processing times were generated randomly from a uniform distribution in the range 1 to 99. Results were based on the average of 100 randomly problem instances. Our observations are as follows:

- ETCBB could readily solve 2000-job problems; for the most restricted case with d=0.9d* and with α: β = 1:5, it took around 23 seconds to solve 2000-job problems.
- DP, A* and DFBB performed poorly in this domain For a node limit of 16k nodes, DP ran out of memory for 20-job problems. A* with heuristic modification (Mero 1984) could solve upto 100-job problems. Other variants of A* with tie resolution in favour of lower or higher g-valued nodes indicated a similar trend. DFBB took 8.28 seconds to solve 100-job problem instances for the most restricted case generating 294463 nodes (Table 3). We could not run DFBB to completion for 200-job problem instances since it took long time to execute.

Node limit =16k nodes				
d=0.9d* α:β=1:5				
Job size	DFBB		A*	
	Node Gen	Time (secs)	Node Gen	Time (secs)
20	300	0.0	251	0.0
40	2968	0.06	877	0.04
60	17258	0.41	3358	0.24
80	20589	0.53	7378	0.80
100	294463	8.28	13023	2.10

Table 3: DFBB and A* for the WET problem

- ETCBB generated 8663 nodes for the most restricted problem which is smaller than the 16k node limit. Hence the node replacement strategy has no role to play. The TCBB version would have performed in a similar way. We have not experimented with the TCBB version because we expect it to perform similarly and secondly, our objective is to show the applicability of the scheme to this domain, not to compare the relative performance of the two versions. Since no nodes were replaced, this remarkable performance of the scheme can thus be attributed to dynamic heuristic improvements incorporated in the algorithm.

We now present the SQP problem and show that ETCBB can optimally solve larger instances than GREC in a given available memory.

Sequence-dependent Quadratic penalty problem (SQP)

The SQP problem is similar in formulation to the QP problem except that jobs may have setup times which are *sequence-dependent*. Setup time s_{ij} for job J_j when it immediately follows job J_i is said to be sequence-dependent if the setup time is dependent on the preceding job J_i. The presence of setup times makes the problem extremely difficult to solve and has been shown to be NP-complete in (Rinooy Kan 1976).

Sen and Bagchi (1996) have shown that due to the presence of sequence-dependent setup times, evaluation function becomes non-order-preserving. As a result, no path can be discarded and A*-based graph search can fail to output optimal solutions. Tree search methods tend to be inefficient because many duplicate nodes get generated. They have suggested a graph search algorithm GREC to optimally solve this problem. GREC has been reported to solve 20-job problems using a hash queue of 200k nodes

We have applied ETCBB to solve this problem optimally. Nodes were represented as pair (V, J_i) where V is the of jobs scheduled and J_i is the last job (Sen and Bagchi 1996). Computation of heuristic estimate and successor ordering are followed as in GREC. Due to the presence of setup times, cost of an arc below a node depends on the incoming path to the node. As a result, cost of a path below a node is also dependent on the incoming path to the node. Hence no node can be labelled SOLVED by the ETCBB algorithm and in our implementation, we have not used the SOLVE labelling procedure.

Node limit =16k nodes				
Job size	GREC		ETCBB	
	Node Gen	Time (secs)	Node Gen	Time (secs)
10	344	0.01	673	0.01
12	920	0.04	2121	0.06
14	2287	0.12	6089	0.20
16			17369	0.71
18			54180	2.75
20			172110	10.64
22			1298847	98.53

Table 4: GREC and ETCBB for the SQP problem with 16k node limit

The use of b-values of nodes in ETCBB helps in outputting optimal solution in a way similar to that of GREC. Since no g-value of nodes are maintained, no paths need to be discarded. When a node n=(V, J_k) first enters the search graph, the completion time t_k of node n's last job is stored at node n as a parameter T. At a subsequent instant when n is reached along a different path with t'_k being the completion time of job J_k, T is reset to t'_k. The b-value b(n) at the node n depends on the incoming path and thus it depends on T. The value of b(n) along the path of downward movement can be expressed as

$$b(n) = AT^2 + BT + C$$

where A, B and C are parameters dependent on node n but independent of T. Let Q' be the remaining jobs to be processed at n. Then

$$A = \Sigma \{ \beta_k \mid J_k \text{ is in } Q' \}$$

$$B = 2 \Sigma \{ \beta_k t'_k \mid J_k \text{ is in } Q' \},$$

and C can be viewed as the b-value at node n with node n as the origin (i.e. with T = 0). The job completion times t'_k in B also take node n as origin. A, B, and C must be stored at each node. Table 4 shows our experimental results and compares the performance of ETCBB with GREC. We can observe the following:

- With 16k node limit, while GREC ran out of space for 16-job problems, ETCBB could solve 22-job problems within 100 seconds. However, GREC ran faster than ETCBB. In (Sen and Bagchi 1996), GREC was shown to solve 20-job problems but using hash queue of size around 200k nodes. We ran both ETCBB and GREC with 200k node limit on a faster computer SUN Enterprise 5500. The results are given in Table 5. GREC ran faster than ETCBB for 20-job problems but ran out of "overflow" area for 22-job problems.

Node limit =200k nodes				
Job size	GREC		ETCBB	
	Node Gen	Time (secs)	Node Gen	Time (secs)
20	36646	6.13	151741	8.59
22			460041	32.57

Table 5: GREC and ETCBB for the SQP problem with 200k node limit

- As in the QP problem, the node generation per second is higher in GREC than ETCBB due to higher overhead of node replacement strategy in ETCBB. Since both the algorithms cache nodes, the better performance of ETCBB is due to its memory management scheme.

Conclusion

This paper shows that the Transpose-and- Cache Branch-and-Bound scheme has a future in solving large instances of problems with natural cut-off bound. Specifically, in some single machine job sequencing problems, the scheme has advanced the existing state of solutions. We hope that this paper would encourage future research in the enhancement and application of the scheme.

Many questions remain. The problems with setup times are difficult and needs special attention. Also, how would the TCBB scheme perform in other domains like project scheduling? Future research should address these issues.

Acknowledgments. This research was partly supported by the Indian Institute of Management Calcutta where both the authors were employed. The authors are grateful to Herman Kaindl for his invaluable suggestions. The authors also thank Gerhard Kainz for sending us the TCBB code.

References

Bagchi, U.; Sullivan, R.; and Chang, Y. 1987. Minimizing Absolute and Squared Deviation of Completion times About a Common Due date. *Naval Research Logistics* 34: 739—751.

Emmons, H. 1987. Scheduling to a Common Due Date on Parallel Uniform Processors. *Naval Research Logistics* 34:803—810.

Hall, N. G.; Kubiak, W.; and Sethi, S. P. 1991. Earliness Tardiness Scheduling Problems, II: Weighted Deviation of Completion Times About a Restrictive Common Due Date. *Operations Research* 39(5):847—856.

Hoogeven, J. A.; Oosterhout, H.; and Van De Velde, S. L. 1994. New Lower and Upper Bounds for Scheduling around a Small Common Due Date. *Operations Research* 42(1):102--110.

Kaindl, H., and Kainz. G. 1997. Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence Research* 7:283—317.

Kaindl, H.; Kainz, G.; Leeb, A.; and Smetana, H. 1995. How to Use Limited Memory in Heuristic Search. In *Proc. Fourteenth International Joint Conference on Artficial Intelligence (IJCAI-95).* 236—242. San Francisco, CA: Morgan Kaufman Publishers.

Mero, L. 1984. A Heuristic Search Algorithm with Modifiable Estimate, *Artificial Intelligence* 23(1): 13—27.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced Iterative deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701—710.

Rinooy Kan, A. H. G. 1976. *Machine Complexity Problems: Classification Complexity and Computations.* Nijhoff, The Hague.

Sen, A. K., and Bagchi, A. 1989. Fast Recursive Formulations for Best-first Search that Allow Controlled Use of Memory, In *Proc. Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89).* 297—302. San Francisco, CA: Morgan Kaufman Publishers.

Sen, A. K., and Bagchi, A. 1996. Graph Search Methods for Non-order-preserving Evaluation Functions: Applications to Job Sequencing Problems. *Artificial Intelligence* 86(1):43—73.

Ventura, J., and Weng, M. X. 1995. An Improved Dynamic Programming Algorithm for the Single-Machine Mean Absolute Deviation Problem with a Restrictive Common Due Date. *Operations Research Letters* 17:149--152.