

Non-Order-Preserving Evaluation Functions: Recursive Graph-Search Methods for Job Sequencing Problems

Anup K. Sen¹ and A. Bagchi¹

School of Industrial Management

New Jersey Institute of Technology

University Heights, Newark, NJ 07102

E-mail: sen@tesla.njit.edu, bagchi@tesla.njit.edu

Abstract

A* sometimes runs significantly faster than a tree-search branch-and-bound procedure if the search graph is represented as a graph and not as a tree. But graph search with A* is suitable only when the evaluation function is order-preserving, *i.e.*, in situations where a path in the explicit graph from the root to a node can be discarded when a better path to the node is found. There exist interesting classes of problems, however, such as one-machine minimum-penalty job sequencing problems with sequence-dependent setup times, for which the evaluation function is non-order-preserving. The search method cannot then discard any paths, and must store the entire explicit graph. GREC, a variant of an earlier recursive memory-constrained algorithm of the authors called MREC, has the desired properties and can be used in place of A*. Experimental results indicate that when the penalty is linear or quadratic, GREC is faster than tree search schemes and is able to solve problem instances of larger size.

1 Introduction

In recent years, various attempts have been made to widen the range of problem areas to which Algorithm A* can be usefully applied. As part of this effort, A* has been employed to solve optimization problems in areas such as project scheduling [Bell and Park 1990] and rectangular stock cutting [Viswanathan and Bagchi 1988 1993]. In both of these cases the search graph is a tree, so there is little distinction between running A* and running a standard best-first branch-and-bound procedure. In some application areas, however, such as in job sequencing and job scheduling [Sen et al. 1991, Ramaswamy and Sen 1992], far higher execution speeds have sometimes been achieved when the search graph has been represented as a graph instead of as a tree. Consider the optimal sequencing of jobs on one machine. It has been reported recently by Sen and Bagchi [1993] that

Townsend's [1978] classic best-first branch-and-bound algorithm for minimal penalty sequencing of jobs when the penalty functions are quadratic runs 27 times faster for 16 jobs when implemented as an A*-based graph search scheme. The existence of such examples suggests that A*-based graph search methods are likely to find greater use in future in the solution of sequencing, scheduling, stock cutting and related problems.

But A* is not suitable for all types of one-machine job sequencing problems. One major difficulty that must be resolved when a job sequencing problem is formulated as an A*-search problem is that the evaluation function is sometimes found to be *non-order-preserving* [Pearl 1984, pp. 100-102]. When the evaluation function is order preserving, a path from the root to a node can be discarded when a path of lower cost has been found. A* is based on this principle. But in the non-order-preserving case, no path can be discarded, because it is possible for a path that is currently unpromising to become promising at later stages of the search process. Suppose we allow jobs to have *setup times*. The setup time for a job is the time taken to get the machine ready for processing after the processing of the previous job has been completed and before the processing of the current job can begin. The setup time for a job J is *sequence independent* if it does not depend on which job is processed immediately prior to job J; otherwise, it is *sequence dependent*. When setup times are ignored or assumed to be sequence independent, A*-based graph search methods yield optimal solutions. When setup times are sequence dependent, however, the evaluation function becomes non-order-preserving. Unfortunately, A* is not at all suited to the use of non-order-preserving evaluation functions, *unless* the search graph happens to be a tree. Graph search can sometimes be faster than tree search, and for a given memory, can solve problem instances of larger size. If these advantages are to be retained, the entire explicit graph needs to be stored, and an alternative algorithm must be found.

The MREC algorithm of Sen and Bagchi [1989] appears to answer the need. MREC is a constrained-memory graph-search algorithm that can be viewed as a generalization of the tree-search scheme IDA* [Korf 1985]. Unlike A*, MREC

¹ On leave from the Indian Institute of Management Calcutta.

is recursive, and it does not use an OPEN list. All it needs is sufficient memory for its implicit stack. But it can be fed at runtime a parameter M which tells it how much memory is available for use. In this memory, MREC stores as much as possible of the explicit graph. When M is large, MREC stores the entire explicit graph, and when M is small, MREC becomes identical with IDA*. IDA* always makes a large number of node expansions. It runs fast on the 15-puzzle because with the Manhattan heuristic the time to expand a node is small. If node expansion time is significant, MREC is to be preferred over IDA*.

In this paper we explain how MREC can be employed to solve search problems with non-order-preserving evaluation functions. We proceed in three steps:

- i) First, we demonstrate the existence of important and interesting real life applications which are inconvenient to solve using graph search methods based on A* because the evaluation functions are non-order-preserving. Tree search would solve the problems, but owing to duplication of nodes would run slowly and expand some nodes many times.
- ii) Secondly, we show that MREC($M = \infty$) can solve such problems. It would seem, therefore, that MREC($M = \infty$) has certain unexpected advantages over A*. Since we are only interested in MREC when M is large, we give a new name to MREC($M = \infty$). We call it GREC. Note that GREC never expands a node more than once.
- iii) Thirdly, we propose new methods of solution for the linear penalty and quadratic penalty versions of the one-machine minimum-penalty job-sequencing problem with sequence-dependent setup times. The quadratic penalty version of the problem has an inherent practical interest, but it is generally considered very difficult to solve, and this is perhaps the first time that a reasonably efficient method of solution is being proposed for it.

In section 2 we review the GREC algorithm, and in section 3 we introduce the one-machine minimum-penalty job-sequencing problem. Section 4 presents our solutions for the linear and quadratic penalty cases. We provide experimental results in support of our claim that GREC runs faster than tree search methods and, for a given memory, solves problems of larger size.

2 Algorithm GREC

GREC is identical to MREC with the memory parameter M set to infinity. It outputs optimal solutions when heuristic estimates are admissible, never expanding a node more than once. Some of its important features are noted below.

- i) GREC is based on the recursive procedure EXPLORE, which, at each iteration, explores the explicit graph below the root node s . Initially the explicit graph contains only the root node. EXPLORE moves down a path in the explicit graph, making recursive calls to itself, until it encounters a *tip node*, i.e., a node which has no successors in the explicit graph. It expands the

```

program GREC;
var terminate;
begin (* initially the explicit graph
      contains the root node  $s$  *)
    terminate := false;
    initialize  $s$ ; (*  $b(s) = 0$  *)
    repeat
        EXPLORE( $s$ )
    until terminate;
    output  $b(s)$  as solution cost;
    output outpath as solution path;
end.

```

```

procedure EXPLORE( $n$ :node);
begin
    if  $n$  is a goal node then
        begin
            terminate := true;
            return;
        end;
    if  $n$  is a tip node then EXPAND( $n$ );
    UPDATE( $n$ );
end;

```

```

procedure EXPAND( $n$ :node);
begin
    for each successor  $n_i$  of  $n$  do
        if  $n_i$  is not present in the explicit graph then
            begin
                initialize  $n_i$ ; (*  $b(n_i) = h(n_i)$  *)
                add  $n_i$  and the edge ( $n, n_i$ ) to the explicit graph;
            end
        else
            add the edge ( $n, n_i$ ) to the explicit graph;
end;

```

```

procedure UPDATE( $n$ :node);
var cutoff : integer;
begin
    cutoff :=  $\infty$  (* a very large value *);
    for each successor  $n_i$  of  $n$  do
        begin
            if  $b(n) \geq b(n_i) + c(n, n_i)$  then
                begin
                     $b(n_i) := b(n) - c(n, n_i)$ ;
                    EXPLORE( $n_i$ );
                    if terminate then
                        begin
                            add  $n_i$  to outpath;
                            return;
                        end;
                end;
            if  $b(n_i) + c(n, n_i) < cutoff$  then
                cutoff :=  $b(n_i) + c(n, n_i)$ ;
        end;
     $b(n) := cutoff$ ;
end.

```

tip node and adds the new nodes and edges to the explicit graph. As in IDA*, a cutoff value is used for monitoring the downward movement. GREC terminates when it encounters a goal node.

ii) Each node n in the explicit graph has a b -value $b(n)$ which stores the current estimate of the cost of a path of least cost from n to a goal node. When n is a tip node, $b(n)$ is set equal to the heuristic estimate $h(n)$. The downward movement along a path is determined by the b -values of the nodes on the path. When GREC on reaching a node n finds a successor n_i of n such that $b(n) \geq b(n_i) + c(n, n_i)$, it explores n_i . If $b(n) > b(n_i) + c(n, n_i)$, then $b(n_i)$ gets reset to a larger value, and this allows a deeper exploration below n . If $b(n) < b(n_i) + c(n, n_i)$ for every successor n_i , then none of the successors get explored. In this case, $b(n)$ gets reset to the minimum of the $b(n_i)$ values. If n is explored at a subsequent instant, n will have a successor n_i for which $b(n) \geq b(n_i) + c(n, n_i)$, and n_i will be explored. The explicit graph thus grows in size, and more and more nodes and edges get added to it.

iii) EXPLORE itself uses two procedures, EXPAND and UPDATE. EXPAND expands a tip node and adds newly generated nodes and edges to the explicit graph. UPDATE explores the graph below a node and updates the b -values of nodes. The algorithm stores the output solution path in *outpath*. The explicit search graph and its associated parameters are assumed to be accessible to all the procedures.

Example 1: For the search graph in Figure 1, we want to find the minimum cost path from the start node s to a goal node. The sequence of node expansions made by GREC is $s \ n \ p \ m$. A*, on the other hand, expands nodes in the sequence $s \ n \ p \ m \ p$, expanding the node p twice. Let each instant correspond to a fresh call to EXPLORE() with argument s ; at the i^{th} instant, EXPLORE(s) is called for the i^{th} time. The explicit graphs at instant 3 and at termination of GREC are shown in Figure 2. In the figures, heuristic estimates and b -values of nodes are encircled. At instant 1, s gets expanded and $b(s)$ gets updated to 6, which is the minimum of the two values $c(s, m) + b(m)$ and $c(s, n) + b(n)$. At instant 2, n gets explored. Since $b(n)$ is 3 and $c(n, p) + b(p)$ is also 3, p gets expanded, and the b -values of p , n and s are revised to 10, 13 and 11 respectively. At instant 3, m gets expanded and $b(s)$ is revised to 12. At instant 4, GREC terminates with $s \ m \ p \ r$ as the minimal cost path. □

Empirical results on a variety of problems indicate that GREC is very fast when evaluation functions are order-preserving. On the 8-puzzle problem with the Manhattan heuristic, it is as fast as IDA*. On the travelling salesman problem using the evaluation function of Little *et al.* [1963], and on tree search problems, it is as fast as A* [Sen and Bagchi 1989]. In problems where a node is expanded more than once and node expansion time is appreciable, it tends to be much faster than A* [Sen 1988]. The power of GREC comes from the use of the b -value, which estimates the cost

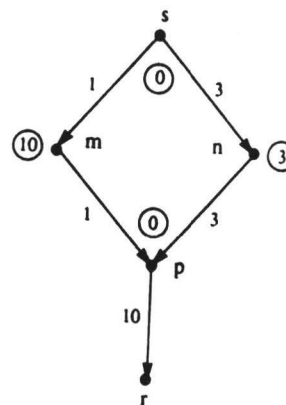


Figure 1: Search Graph

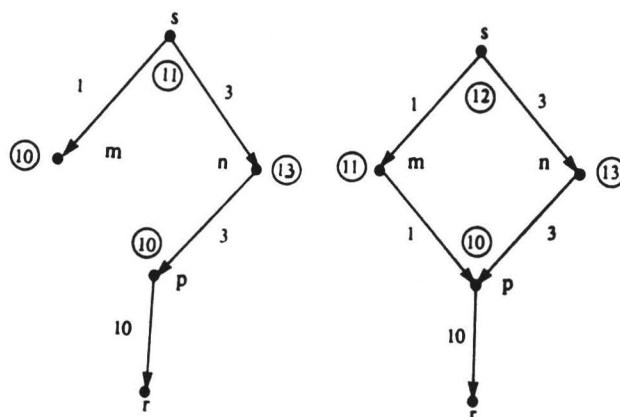


Figure 2: Explicit Graphs During the Execution of GREC

of the best path from the node to a goal; the b -value helps to determine whether further exploration below the node is worth pursuing, and is an extremely useful feature.

Comparing the properties of GREC, A* and IDA*, we find that:

1. The sequences in which nodes are expanded by GREC and by A* are not the same.
2. Unlike A*, GREC never expands a node more than once.
3. When the heuristic estimate function is admissible, GREC always finds a minimal cost solution path. A* may not find an optimal path for non-order-preserving evaluation functions. Even for order-preserving evaluation functions, the optimal solution paths found by the two algorithms can differ. However, GREC and IDA* always output the same solution path.
4. The worst-case running time of GREC, like that of A* and IDA*, can be exponential in the number of nodes in the search graph.

3 One-Machine Job-Sequencing Problems

A typical one-machine job sequencing problem has the following form. Jobs J_i with processing times a_i , $1 \leq i \leq N$, are submitted to a one-machine job-shop at time $t = 0$. The jobs are to be processed on the given machine one at a time. Let the processing of job J_i be completed at time t_i . *Penalty functions* $G_i(\cdot)$, $1 \leq i \leq N$, are supplied, such that the penalty associated with completing job J_i at time t_i is $G_i(t_i)$. An example of a penalty function is $G_i(t_i) = p_i t_i^2$, where the p_i 's are given constants, the penalty associated with a job being proportional to the square of the completion time of the job. The jobs must be sequenced on the machine in such a way that the total penalty

$$F = \sum \{ G_i(t_i) \mid 1 \leq i \leq N \}$$

is minimized. The penalty functions are non-decreasing, and in general non-linear. Problems of this type have obvious relevance to industry, but are known to be hard to solve. In a more general setting, jobs have *setup times*. The setup time for a job is the time taken to get the machine ready for processing after the processing of the previous job has been completed and before the processing of the current job can begin. If J_i is the first job in the sequence, then its setup time is $s_{0,i}$. The setup time for job J_j when it immediately follows job J_i in the processing sequence is $s_{i,j}$. After the last job in the sequence is completed, the machine need not be brought back to any specific state. Setup times are said to be *sequence independent* (or *separable* or *additive*) if $s_{i,j}$ can be expressed as a sum of two terms, one of which depends only on job J_i and the other only on job J_j ; otherwise, the setup times are *sequence dependent* [French 1982, Bianco *et al.* 1988]. Branch-and-bound approaches frequently make the simplifying assumption that setup times are sequence independent [Mason and Anderson 1991]. Solution methods that work in the absence of setup times can frequently be extended to the situation when setup times are sequence independent. But if we make the more realistic assumption that setup times are sequence dependent, entirely new solution methods are typically required.

In the absence of setup times, A^* solves many of the one-machine job sequencing problems efficiently. Let there be N jobs to be sequenced. Nodes in the search graph correspond to unordered subsets of jobs; the root node is the empty set. When a node corresponding to a subset of k jobs gets selected from OPEN and expanded, it generates $N-k$ sons, each son being a subset of $k+1$ jobs. An edge signifies that one more job has been processed, and the cost of the edge is the penalty associated with that job. A problem gets solved when a complete set of N jobs gets selected from OPEN. The method of computation of the heuristic estimate at non-goal nodes varies from problem to problem. In the sequence-independent quadratic penalty case, for instance, a very fast and memory-efficient implementation results if Townsend's [1978] method of computing the evaluation function is adopted [Sen and Bagchi 1993].

When setup times are sequence dependent, a different

kind of search graph is needed. Nodes can no longer correspond just to subsets of jobs, because the last job that has been processed must be remembered. Thus a node now corresponds to an ordered pair, where the first component is the job subset, and the second component is that job in the subset that has been processed last. As before, the cost of an edge is the penalty associated with the corresponding job. A node n in the search graph has the form (S, J_i) , where S is a subset of jobs and J_i is the last job processed from among the jobs in S . Let P_1 and P_2 be two paths from the root node s to n . The jobs processed along either of the two paths all belong to S . Let t_i be the time at which J_i completes processing when the jobs in S are processed in the sequence determined by path P_1 , and let t'_i be the time at which J_i completes processing when the jobs are processed in the sequence determined by path P_2 . When setup times are ignored or are assumed to be sequence independent, t_i always equals t'_i . But when setup times are sequence dependent, t_i may not equal t'_i , because the setup times of jobs along the two paths are not identical. It can happen that the cost associated with P_1 is less than the cost associated with P_2 , but $t_i > t'_i$. Since the penalty corresponding to a job depends on its time of completion, the arc costs of the arcs emanating from node n as well as the heuristic estimate at node n depend in general on the path by which n is reached from the root, *i.e.*, the arc costs and the heuristic are *path-dependent* [Bagchi and Mahanti 1983]. It is also possible for a complete job sequence of N jobs that is an extension of the job subsequence determined by P_2 to have a lower total cost than a job sequence of N jobs that is an extension of

Table 1: Job Setup and Processing Times

Job	Setup Times				Proc. Times
	1	2	3	4	
1	-	1	1	∞	1
2	1	-	3	∞	4
3	∞	∞	-	10	3
4	∞	∞	∞	-	10

P_1 . But a normal implementation of A^* would discard path P_2 at node n because its cost is higher than that of P_1 , and job sequences that are extensions of the sequence determined by P_2 would not subsequently get considered by the algorithm. In the linear penalty case, it is possible to skirt this difficulty by making a minor modification in A^* [Ramaswamy and Sen 1992]; but the method does not seem to work in the quadratic penalty case.

Example 2: Consider the following linear penalty problem with 4 jobs. Let the penalty coefficients be unity for all the jobs; then the penalty for a job J_i equals t_i , where t_i is the finish time of the job, and the total penalty is the sum of the

individual job penalties. Table 1 gives the setup times and the processing times. Setup time $s_{i,j}$ is the entry in the i th row and j th column of the table. The initial setup times $s_{0,j}$ are assumed zero. The cost of the arc (m,r) between node $m = (\{1,2,3\}, 3)$ and the goal node $r = (\{1,2,3,4\}, 4)$ depends on the path by which we reach m from the root. If we process J_1 before J_2 then the cost of the path P_1 from the root to m is 19, and $c(m,r) = 32$. If we process J_2 before J_1 then the cost of the path P_2 from the root to m is 20, and $c(m,r) = 30$. If at m we favor P_1 and discard P_2 , then we end up giving preference to the inferior of the two solution paths. The minimum penalty job sequence in this case happens to be $(J_2 J_1 J_3 J_4)$. $\square$

4 Non-Order-Preserving Evaluation Functions

4.1 Linear Penalties

When penalties are linear and setup times are sequence dependent, a modified version of A^* finds minimum penalty sequences [Ramaswamy and Sen 1992]. The same problem can be solved by GREC in the following way. When a node $n = (S, J_i)$ first enters the explicit graph as a result of the expansion of its parent node, the completion time t_i of n 's last job J_i is saved in a parameter T associated with n . At a subsequent instant, if the EXPLORE procedure happens to reach n again, the completion time of job J_i has a value t'_i which is in general different from t_i because there are many paths from the root node to n . T must then be reset to t'_i .

Table 2: Linear Penalty Functions

Algorithm	No. of jobs	Exec. time (secs)	Nodes Gen.	Nodes Exp.
A^* (Tree)	12	0.04	710	93
	14	0.13	1952	217
	16	0.39	4988	481
	18	1.33	14039	1182
	20	4.55	36818	2818
GREC	12	0.02	333	64
	14	0.06	657	119
	16	0.17	1329	225
	18	0.43	2563	415
	20	1.21	5400	842
	22	6.31	10475	1634
	24	35.36	18965	2836

The value of $b(n)$ depends on the path by which n is reached from the root, and thus T plays a role in the computation of $b(n)$. When T changes in value from t_i to t'_i , $b(n)$ changes in value by the amount $(t'_i - t_i) \sum c_k$, where we sum over those jobs which remain to be processed at n .

Example 3: Consider the job sequencing problem of Example 2. Let us assume for simplicity that all nodes have zero heuristic estimates. GREC outputs the minimum penalty sequence $(J_2 J_1 J_3 J_4)$ with cost equal to 50 without expanding any node more than once. The algorithm operates as described in Example 1. Suppose a node n has been reached from the root at some instant along a path P_1 . If n is reached from the root at a subsequent instant along another path P_2 , arc costs and b -values at n are recomputed with the help of the stored value of T . $\square$

In our implementation, heuristic estimates at tip nodes were computed as in [Ramaswamy and Sen 1992]. Let $n = (S, J_i)$ be a tip node in the explicit graph. Let Q be the set of jobs yet to be processed at n . For a job J_k in Q , take the effective setup time s_k to be $\min\{s_{jk}\}$, where the minimum is taken over all jobs J_j in $Q \cup \{J_i\}$. Let us call the $(s_k + a_k)$ values the *effective* job processing times. Order the jobs J_k in Q in non-decreasing order of $(s_k + a_k)/p_k$, compute the finish times t_k of the jobs, and let the heuristic estimate be $\sum p_k t_k$, where the summation is over all jobs in Q . This yields a very good heuristic, as our experimental results in Table 2 show.

In the experiments, the programs were written in C and run on a UNIX-based DEC 5900 system. Processing times of jobs were integers and were chosen randomly in the range 1 to 99 from a uniform distribution. Penalty coefficients and setup times were integers in the range 1 to 9 and were generated in the same way. The initial setup times $s_{0,j}$ were assumed zero for simplicity. For a given number of jobs, the execution time, the total number of nodes generated, and the total number of nodes expanded were averaged over 100 runs. The A^* tree implementation was run upto 20 jobs. GREC was able to solve larger problems, and was about 3.76 times faster for 20 jobs than the A^* tree procedure. Every effort was made to ensure that the implementations were as efficient as possible. The OPEN set of A^* was maintained as a priority queue. In GREC a hashing scheme was used for checking whether a newly generated successor was already present in the explicit graph.

4.2 Quadratic Penalties

The elegant and efficient A^* graph implementation of the quadratic penalty case that is realized using Townsend's [1978] method for computing the evaluation function does not appear to permit a generalization to sequence dependent setup times. To resolve this difficulty, GREC must be used in place of A^* . The method of solution is similar to that in section 4.1; some additional computations are needed at each step because the penalty function is non-linear.

At a tip node n , the effective processing time $(s_k + a_k)$ for

a job J_k in Q is computed as before. A heuristic estimate is then found for n by a modification of Townsend's [1978] method. The computation procedure for the heuristic is described in [Sen and Bagchi 1993]. In the UPDATE procedure, the b -value at a node n along the path of downward movement can be expressed as

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

where T , A , B and C are parameters that depend on node n . A , B and C do not depend on T . We have

$$A = \sum \{ p_k \mid J_k \text{ is in } Q' \}$$

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

and C can be viewed as the b -value at node n , where job completion times in B and C take n as the origin. A , B , and C must be stored at each node and get reset in UPDATE.

Table 3: Quadratic Penalty Functions

Algorithm	No. of jobs	Exec. Time (secs)	Nodes Gen.	Nodes Exp.
A* (Tree)	12	0.15	3479	462
	14	0.69	12857	1441
	16	3.38	49923	4957
GREC	12	0.08	856	214
	14	0.33	2159	512
	16	1.32	5067	1169
	18	12.16	12932	2954
	20	82.95	31750	7042

Experimental results are shown in Table 3. The tree implementation of A^* also used the Townsend heuristic. GREC could be run for a larger number of jobs, and was about 2.56 times faster than A^* tree search for 16 jobs. For 18 jobs, the tree search scheme generated more than 450,000 nodes in some instances. Since the number of computations at each step in the quadratic case is greater than in the linear case, running times in Table 3 are higher than in Table 2.

References

- [Bagchi and Mahanti 1983] A. Bagchi and A. Mahanti, Search Algorithms Under Different Kinds of Heuristics - A Comparative Study, *JACM*, vol. 30, no. 1, January 1983, pp. 1-21.
- [Bell and Park 1990] C. E. Bell and K. Park, Solving Resource-Constrained Project Scheduling Problems by A^* Search, *Naval Research Logistics*, vol. 37, 1990, pp. 61-84.
- [Bianco et al. 1988] L. Bianco, S. Ricciardelli, G. Rinaldi and A. Sassano, Scheduling Tasks with Sequence Dependent Processing Times, *Naval Research Logistics*, vol. 35, no. 2, 1988, pp. 177-184.
- [French 1982] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood Ltd., 1982
- [Korf 1985] Richard E. Korf, Depth-First Iterative Deepening : An Optimal Admissible Search, *Artificial Intelligence*, vol. 27, no. 1, 1985, pp. 97-109.
- [Little et al. 1963] J. D. C. Little, K. G. Murty, D. W. Sweeney and G. Karel, An Algorithm for the Traveling Salesman Problem, *Operations Research*, vol. 11, 1963, pp. 972-989.
- [Mason and Anderson 1991] A. J. Mason and E. J. Anderson, Minimizing Flow Time on a Single Machine with Job Classes and Setup Times, *Naval Research Logistics*, vol. 38, no. 3, 1991, pp. 333-350.
- [Pearl 1984] J. Pearl, *Heuristics : Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, Mass., 1984.
- [Ramaswamy and Sen 1992] Ramkumar Ramaswamy and Anup K. Sen, Single Machine Scheduling as a Graph Search Problem With Path-Dependent Arc Costs, *Proc. ECAI-92*, European Conference on Artificial Intelligence, Vienna, August 1992, pp. 11-15.
- [Sen 1988] Anup K. Sen, *Heuristic Search Algorithms : A Theoretical-cum-Empirical Evaluation*, Unpublished Ph.D. Thesis, Computer Science, University of Calcutta, 1988.
- [Sen and Bagchi 1989] Anup K. Sen and A. Bagchi, Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory, *Proc. IJCAI-89*, Detroit, August 1989, pp. 297-302.
- [Sen and Bagchi 1993] Anup K. Sen and A. Bagchi, Job Sequencing with Quadratic Penalties: An A^* -Based Graph Search Approach, *Proc. CAIA-93*, Conference on Artificial Intelligence for Applications, Orlando, March 1993, pp. 190-196.
- [Sen et al. 1991] Anup K. Sen, A. Bagchi and Bani K. Sinha, Admissible Search Methods for Minimum Penalty Sequencing of Jobs with Setup Times on One and Two Machines, *Proc. IJCAI-91*, Sydney, August 1991, pp. 178-183.
- [Townsend 1978] W. Townsend, The Single Machine Problem With Quadratic Penalty Function of Completion Times : A Branch and Bound Solution, *Management Science*, vol. 24, no. 5, 1978, pp. 530-534.
- [Viswanathan and Bagchi 1988] K. V. Viswanathan and A. Bagchi, An Exact Best-First Search Procedure for the Constrained Rectangular Guillotine Knapsack Problem, *Proc. AAAI-88*, American Association for Artificial Intelligence, St. Paul, August 1988, pp. 145-149.
- [Viswanathan and Bagchi 1993] K. V. Viswanathan and A. Bagchi, Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems, *Operations Research*, May-June 1993, forthcoming.