ELSEVIER Artificial Intelligence 86 (1996) 43–73 Artificial Intelligence

Graph search methods for non-order-preserving evaluation functions: applications to job sequencing problems[☆]

Anup K. Sen[*], Amitava Bagchi[1]

Indian Institute of Management Calcutta, Joka, D. H. Road, P.O. Box No. 16757, Calcutta 700 027, India

Received March 1994; revised April 1995

Abstract

Graph search with A* is frequently faster than tree search. But A* graph search operates correctly only when the evaluation function is order-preserving. In the non-order-preserving case, no paths can be discarded and the entire explicit graph must be stored in memory. Such situations arise in one-machine minimum penalty job sequencing problems when setup times are sequence dependent. GREC, the unlimited memory version of a memory-constrained search algorithm of the authors called MREC, has a clear advantage over A* in that it is able to find optimal solutions to such problems. At the same time, it is as efficient as A* in solving graph search problems with order-preserving evaluation functions. Experimental results indicate that in the non-order-preserving case, GREC is faster than both best-first and depth-first tree search, and can solve problem instances of larger size than best-first tree search.

1. Introduction

Many attempts have been made in recent years to widen the range of problem areas to which algorithm A* [8] can be usefully applied. As part of this effort, A* has been employed to solve optimization problems in areas such as project scheduling [3] and rectangular stock cutting [29]. In both 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 job sequencing and job

[☆] A major part of the work reported here was completed at the School of Industrial Management, New Jersey Institute of Technology, Newark, NJ 07102, USA, where both authors were visitors.
[*] Corresponding author. E-mail: sen@iimcal.ernet.in.
[1] E-mail: bagchi@iimcal.ernet.in.

scheduling, however, far higher execution speeds have sometimes been achieved with the search graph represented as a graph instead of as a tree [15,22,24,25]. Consider the optimal sequencing of jobs on one machine. It has been found that Townsend's classic best-first branch-and-bound algorithm [28] for minimum penalty sequencing of jobs when the penalty functions are quadratic runs 25 to 40 times faster for 16 jobs when implemented as an A*-based graph search scheme. Such examples suggest that graph search methods are likely to find greater use in future in sequencing, scheduling, stock cutting and related optimization problems.

A* is not the only graph search procedure in existence, however. Some other methods have been proposed in recent years which either view the search space as a graph [9] or try to avoid duplication of nodes in tree search [17,27]. MREC [21] is one of the earliest such methods. It is a constrained memory graph search algorithm that can be viewed as a generalization of the iterative deepening tree search scheme IDA* [10]. Unlike A*, MREC is recursive and does not use an OPEN list. All it needs is sufficient memory for its implicit stack. But it can be fed at run time a parameter M which tells it how much additional memory is available for use. In this memory, it stores as much as possible of the explicit graph. When M is small it approximates to IDA* and expands each node many times. As M increases, MREC makes fewer and fewer node expansions, which is of advantage when node expansion time is significant. When M is large MREC stores the entire explicit graph and does not expand any node more than once; it then becomes comparable in performance to A*.

MREC($M = \infty$), which is called GREC in this paper, is efficient in solving graph search problems with order-preserving evaluation functions. It is also able to find optimal solutions to problems involving *non-order-preserving* evaluation functions (Pearl [14, pp. 100–103]), so in this respect it has a clear advantage over A*. We recall that when an evaluation function is order-preserving, a path from the root to a node can be discarded when a path of lower cost is 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. Such situations arise in job sequencing when jobs are allowed 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 matter 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, and A* is unsuitable unless the search graph is implemented as a tree. To retain the advantages of graph search over tree search for problems of small and medium size, the entire explicit graph has to be stored, and an algorithm like GREC must be used in place of A* [23].

Job sequencing and job scheduling are areas in which it is difficult to find exact solutions since most problems are NP-complete [19]; as a result, approximation methods have been extensively used in recent years. Promising approaches employ artificial neural networks and tabu search [16]. It has been demonstrated, for example, that in flow shop scheduling, tabu search obtains solutions uniformly better than the best of the classical heuristics [26]. A list of applications of tabu search to sequencing and scheduling is given in [16, Table 3.1, p. 128]. Similar approaches might prove useful in minimum penalty job sequencing if only approximate solutions are desired. It is true that in situations that arise in practice, problem sizes tend to be large and exact methods become infeasible. Our interest in this paper, however, is primarily to extend the scope of use of exact methods in job sequencing, so we do not discuss approximation methods in any detail.

This paper is organized as follows:

(a) In Section 2 we review algorithm GREC and explain its principle of operation with the help of an example. We then formally state and prove some of its important properties.

(b) We then try to show how GREC solves one-machine job sequencing problems, our particular interest being in cases that involve sequence-dependent setup times. As a first step in this direction, job sequencing problems are described in Section 3, and the existence of an interesting class of real life problems that have non-order-preserving evaluation functions is demonstrated. Tree search can solve such problems, but owing to duplication of nodes the execution is slow and some nodes get expanded multiple times. Section 4 explains how graph search with GREC can provide answers to such problems. The quadratic penalty version of the one-machine minimum penalty job sequencing problem with sequence-dependent setup times has great practical interest. Depth-first branch-and-bound methods have been employed in the past to solve similar problems [4,13]. We show that when problem instances are of small and medium size, GREC is to be preferred over other methods since it runs faster than both best-first and depth-first tree search, and also solves problems of larger size than best-first tree search. The last section summarizes the paper and compares GREC with some other similar search schemes.

2. Algorithm GREC

GREC is identical to MREC [21] with the memory parameter M set to infinity. It outputs optimal solutions when heuristic estimates are admissible, never expanding a node more than once. We explain its principle of operation below.

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 until it encounters a *tip node*, i.e., a node which as no successors in the explicit graph. It expands the 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 goal node.

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, $b(n)$ equals the heuristic estimate $h(n)$.

```
program GREC;
var terminate;
begin (* initially the explicit graph contains the root node $s$ *)
  terminate := false;
  initialize $s$ (* $b(s) = h(s)$ *);
  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($n$);
    (* updation takes place whether $n$ is a tip node or a non-tip node *)
end;

procedure EXPAND($n$:node);
begin
  (* if $n$ has no successors then $b(n) = h(n) = \infty$, so EXPLORE and therefore
     EXPAND will never get called at $n$ *)
  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 := ∞ (* 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($n_i$); (* exploration continues to greater depths until the bound
                           is exceeded *)
        if terminate then
        begin
          add $n_i$ to outpath; return;
        end;
      end;
      (* at this point $b(n_i) + c(n, n_i) > b(n)$ *)
      if $b(n_i) + c(n, n_i) <$ cutoff then cutoff := $b(n_i) + c(n, n_i)$;
    end;
    $b(n)$ := cutoff;
end.
```

The downward movement along a path is determined by the b-values of the nodes on the path and the costs of the arcs. Let $c(n, n_i)$ denote the cost of the arc (n, n_i). 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_i. If $b(n) < b(n_i) + c(n, n_i)$ for every successor n_i of n, 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 iteration, n will have a successor n_i in the explicit graph for which $b(n) \geq b(n_i) + c(n, n_i)$, and n_i will also be explored.

EXPLORE calls 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; it carries out the exploration by calling EXPLORE, and this makes GREC recursive. The output solution path is stored in *outpath*. The explicit search graph and its associated parameters are assumed to be accessible to all the procedures.

From the above description it is clear that in spite of certain superficial resemblances, GREC differs radically from IDA* in two major respects:

(i) GREC stores the entire explicit graph in memory. This amounts to storing the nodes and the successor lists. IDA*, in contrast, does not store any part of the explicit graph in memory. Because the entire explicit graph is available to GREC, no node needs to be expanded more than once. Note that unlike GREC, A* does not store the entire explicit graph in memory; it stores a spanning tree of the explicit graph that contains the currently known minimum cost path to each node.

(ii) IDA* is a tree search algorithm. When it searches a graph that is not a tree, it implicitly converts the graph to a tree in the course of the search. GREC

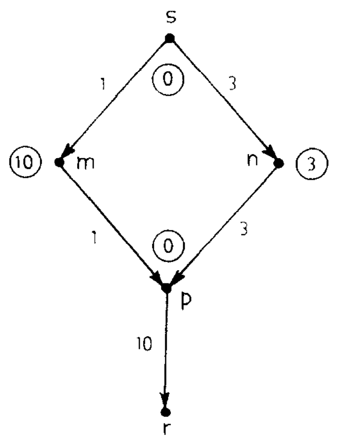

Fig. 1. Search graph for Example 1.

is a genuine graph search method and views a graph as a graph; it does not convert the graph to a tree.

Example 1. Consider the search graph shown in Fig. 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 *snpm*, which differs from the sequence *snpmp* of A*. Let each *instant* correspond to a fresh call to EXPLORE(s); at the ith instant, EXPLORE(s) is called for the ith time. The explicit graphs at instant 3 and at termination of GREC are shown in Fig. 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 expanded. Since $b(n)$ is 3 and $c(n, p) + b(p)$ is

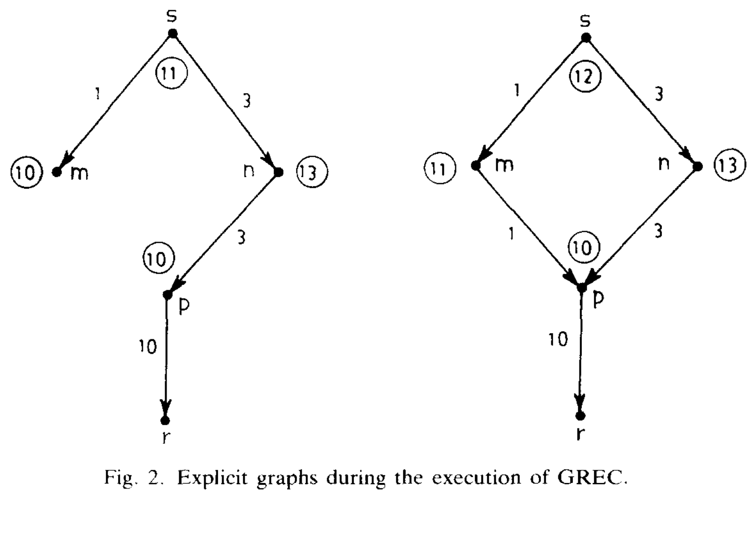

Fig. 2. Explicit graphs during the execution of GREC.

also 3, p also 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 *smpr* as the minimum cost solution path.

An experimental comparison of GREC with A* and IDA* on a variety of problems is given in [21]. The data indicates that GREC runs quite fast in general; a few examples are given below.

(i) On the 8-puzzle problem with the Manhattan heuristic, it is as fast as IDA*. On the corresponding 15-puzzle problem it runs short of memory just like A*. But MREC with limited memory is able to solve this problem as fast as IDA*.

(ii) On the travelling salesman problem using the evaluation function of Little et al. [12], as well as on a graph formulation of the problem suggested by Pearl [14], it is as fast as A*.

(iii) On the rectangular cutting stock problem [29], it is as fast as A*.

(iv) On a general uniform d-ary tree with bi-directional arcs of unit cost, a single goal node, and randomly generated admissible heuristic estimates, it is faster than both A* and IDA*.

The power of GREC comes from the use of the b-value, which estimates the cost 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 characteristics of GREC, A* and IDA*, we find that:

(i) When the evaluation function is admissible and order-preserving, all three algorithms find optimal solutions. A tree search method like IDA* always finds optimal solutions, even when the evaluation function is non-order-preserving, but A*-based graph search methods can fail to do so. GREC, however, is able to solve the problem.

(ii) GREC and A* sometimes expand nodes in different order, and the solution paths can also differ. But GREC and IDA* always output the same solution path.

(iii) Unlike A* and IDA*, GREC never expands a node more than once.

(iv) 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.

Some of these issues, as for example (iv), are discussed in greater detail in Section 2.1.1.

2.1. *Properties of GREC: theoretical formulation*

2.1.1. *Constant arc costs*

We now state and derive some important theoretical properties of GREC. We initially assume that arc costs are constant, and later extend the results to path dependent arc costs. The notation and terminology are adapted from [1]. Proofs of important results are given in Appendix A.

A *search graph* G is a directed graph with a special node s called the *root* node, and a non-empty set of *goal* nodes. Let $r, r_1, r_2, \ldots$ be the goal nodes in G, and $m, n, \ldots$ the other nodes in G. Each directed arc (m, n) has a finite, strictly positive arc cost $c(m, n)$. A *path* is a finite sequence of directed arcs; a *solution path* is a path from the root node to a goal node. The *cost* $c(P)$ of a path P is the sum of the costs of the arcs which make up the path. Our objective is to find a solution path of minimum cost in G. GREC, like A*, tries to find this path by systematically searching G. Each node m in G has an associated non-negative *heuristic estimate* $h(m)$, and the search is guided by these heuristic estimate values. The search graph G is called the *implicit graph* and is not really supplied to the algorithm. What is given is a set of rules for generating the *explicit graph*, which is a subgraph of G. Initially the explicit graph consists of the root node s. When s is expanded its successors are added to the explicit graph. At subsequent instants, fresh nodes and arcs get included in the explicit graph as more and more nodes get expanded.

The following conditions are frequently imposed on the search graph G to ensure that the search is well defined and that it terminates successfully:

(i) G has exactly one root node, at least one goal node, and at least one solution path. G can have infinitely many nodes and arcs, but each node in G has finitely many immediate successors. It is permissible for G to have directed loops or cycles.

(ii) There is a real number $\tau > 0$ such that for each arc (m, n) in G, $c(m, n) \geqslant \tau$. Since $c(m, n) > 0$, this condition always holds if G has finitely many nodes, and is therefore always true in practical situations. But in theoretical studies, when G has infinitely many nodes and arcs, in order to ensure that the search algorithm terminates a non-zero lower bound must be imposed on the arc costs.

(iii) For each non-goal node m in G, the heuristic estimate $h(m)$ is finite if there is a path in G from m to a goal node. Otherwise, $h(m)$ can be finite or infinite; in particular, we take $h(m)$ to be infinite if m is a non-goal node with no successors. If r is a goal node then we take $h(r) = 0$.

Definition 2.

(i) Let an *instant* correspond to a fresh invocation of EXPLORE(s); the ith instant is thus the moment at which EXPLORE(s) is called for the ith time. If EXPLORE(n) is invoked at instant j for some node n in the explicit graph, we say that n is *explored* at instant j.

(ii) Let G_j' be the explicit graph with root s at instant j, and $G_j'(m)$ the subgraph of G_j' with m as root. (Consider the moment of time immediately prior to the jth invocation of EXPLORE(s), and look at the explicit graph at that moment.)

(iii) By $b_j(m)$ we mean the b-value of the node m at instant j, and by b_j we mean $b_j(s)$. Let b^* denote the cost of the solution path of minimum cost in G.

(iv) The explicit graph G_j' at instant j is consistent if there is a path P from s to a tip node n in G_j' such that for *every* non-tip node m lying on P, $b_j(m)$

is finite and equals $c(P, m, n) + h(n)$, where $c(P, m, n)$ is the cost of path P from m to n. The path from m to n along P is called the *potential solution path* (psp) below m at instant j; psp (without qualification) refers to the potential solution path below s. At the initial instant, the explicit graph G'_1 contains just the node s. G'_1 is viewed as being consistent since $b(s) = h(s)$, the psp consisting of just the node s. When G'_j is consistent, the tip node n of the psp (where n could be a goal node) gets explored at instant j; if G'_j is not consistent then no tip node is expanded at instant j.

(v) Let $P'_1, P'_2, \ldots, P'_k$ be the paths in G'_j from a node m to tip nodes. Define

$$Q'_j(m) = \min_{1 \le i \le k} \left[\max_{m' \in P'_i} \{c(P'_i, m') + h(m')\} \right].$$

Here $c(P'_i, m')$ indicates the cost of path P'_i from m to m'; $c(P'_i, m)$ is taken to be zero, so that $Q'_j(m)$ equals $h(m)$ if m is a tip node. By Q'_j we mean $Q'_j(s)$. Note that $Q'_1 = h(s)$.

(vi) Let $P_1, P_2, \ldots, P_{k'}$ be the paths in G from m to goal nodes. Define

$$Q(m) = \min_{1 \le i \le k'} \left[\max_{m' \in P_i} \{c(P_i, m') + h(m')\} \right],$$

where $c(P_i, m')$ indicates the cost of the path P_i from m to m'. If m is a goal node then $Q(m) = 0$. If there are no solution paths passing through m then $Q(m) = \infty$. By Q we mean $Q(s)$. Note that Q is always finite.

(vii) The heuristic estimate function is *admissible* if for each non-goal node m in G, $h(m)$ never exceeds the cost of a minimum cost path from m to a goal node; otherwise the heuristic is *inadmissible*. When the heuristic is admissible, $Q = b^*$.

GREC has the property that at termination, the cost of the output solution path is bounded above by Q. When the heuristic estimate function is admissible, this path is the minimum cost solution path in the search graph. The results are stated in Theorems 10 and 12. In order to establish the theorems, we first show that at any node m in the explicit graph, $Q'(m)$ is bounded above by $Q(m)$. As the execution of GREC proceeds, Q'_j approximates Q more closely and finally equals Q.

Lemma 3. *Let m be a node with immediate successors $m_1, m_2, \ldots, m_k$ in G'_j. Then*

$$Q'_j(m) = \max_{1 \le i \le k} [h(m), \min\{c(m, m_i) + Q'_j(m_i)\}].$$

Lemma 4. *Let j and j' be two instants during the execution of GREC, where $j < j'$. Then for any node m in G'_j,*

$$Q'_j(m) \le Q'_{j'}(m) \le Q(m).$$

Lemmas 3 and 4 follow immediately from the definitions of $Q'_j(m)$ and $Q(m)$.

Corollary 5. *Let j and j' be two instants during the execution of GREC, where $j<j'$. Then*

$$Q'_j \leqslant Q'_{j'} \leqslant Q\ .$$

How is the b-value at the root node related to its Q-value? It is bounded above by Q at every instant, as the next lemma shows.

Lemma 6. *Let j be any instant during the execution of GREC. Then*
(i) *$b_j(m) \leqslant Q'_j(m)$ for any node m in G'_j;*
(ii) *$b_j \leqslant Q'_j \leqslant Q$.*

Corollary 7. *If the explicit graph is consistent at instant j, then for every node m on P where P is the psp at instant j, $b_j(m) = Q'_j(m)$.*

Remark 8.
(i) If the heuristic is inadmissible, then at termination there might be a node m on the output path, where m is not the root, for which the b-value is greater than $Q(m)$. See proof of Lemma 6, Appendix A.
(ii) If the search graph is a tree, then at every instant j, $b_j(m) = Q'_j(m)$ for every node m in G'_j.
(iii) When the search graph is not a tree, there are instants at which the explicit graph is not consistent. For example, suppose that in the search graph of Fig. 1, we make $c(m, p) = 6$ and $h(p) = 6$, keeping all other values the same. Then the values taken by $b(s)$ at successive instants are 0, 6, 11, 12, 13, 16, and at instant 5 the explicit graph is not consistent.

Lemma 9. *Let j be any positive integer. If GREC has not terminated by instant j then at some instant $j' \geqslant j$ prior to termination, $G'_{j'}$ is consistent.*

Theorem 10. *Algorithm GREC terminates successfully, i.e., it finds a goal node and outputs a solution path.*

Definition 11. Let b_{GREC} be the b-value at the root s when GREC terminates.

Theorem 12.
(i) *$b_{\text{GREC}} = Q$.*
(ii) *$b_{\text{GREC}} = b^*$ if the heuristic estimate function is admissible.*

Which nodes of the search graph are expanded by GREC? These are the nodes contained in the set V defined below.

Definition 13. We construct a finite set of nodes V as follows:
(i) s is in V.

(ii) A node m of G is in V if there is a path P from s to m such that $c(P) + h(m) \leqslant Q$, and the immediate predecessor of m on P is in V.

Theorem 14. *All nodes expanded by GREC belong to the set V.*

To form an idea about the running time of GREC, it seems appropriate to count the number of times the procedure EXPLORE is called before the algorithm terminates. When EXPLORE(m) is called for the first time, node m gets expanded if it is not a goal node, and the minimum of the values $b(m_i) + c(m, m_i)$ is found over the immediate successors of m. When m is explored subsequently it is not expanded again but the minimum is recomputed. Thus, the number of such computations can serve as a good measure of the running time of GREC. (This makes sense only when G is loop free; when there are loops, the number of revolutions made around a loop depends on the costs of the arcs forming the loop and can be increased by reducing the arc costs.) Do there exist search graphs which force GREC to make an exponential number of calls to EXPLORE? The following example demonstrates the existence of such a search graph.

Example 15. Consider the search graph G shown in Fig. 3. G has the following features:

(i) Total number of nodes $U_1 = 2k + 1$.
(ii) All heuristic estimates are zero.
(iii) The arc costs are as follows:

$$c(n_{i-1}, m_i) = c(m_i, n_i) = 2 \cdot 3^{i-1}, \quad 1 \leqslant i \leqslant k,$$
$$c(n_{i-1}, n_i) = 3^i, \quad 1 \leqslant i \leqslant k.$$

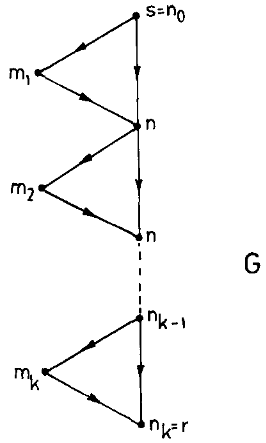

Fig. 3. Search graph for Example 15.

When $k = 4$, GREC expands nodes in the order $n_0m_1n_1m_2n_2m_3n_3m_4$. When n_1 or m_2 gets expanded, there is an inconsistency at m_1 which must be resolved. When n_3 or m_4 gets expanded, inconsistencies must be resolved at the nodes m_1, m_2, m_1, m_3, m_1, m_2, m_1 in the given order. The total number of calls to EXPLORE, which is bounded below by the total number of inconsistencies to be resolved, will be exponential in U_1 for networks of this type.

2.1.2. Path dependent arc costs

We now extend our results to path dependent arc costs. Here the heuristic estimate at a node m and the costs of the arcs emanating from m depend on the path by which m is reached from the root. We redefine some the notation of Section 2.1.1; any notation not specifically redefined continues to have its earlier meaning.

Definition 16.

(i) Let P be a path from the root s to a node m in the search graph G. Let $c(P, m, n)$ be the cost along path P from node m to node n. Let $h(m, P)$ be the heuristic estimate of node m calculated with reference to path P. The heuristic estimate function is *admissible* if for each node m in G and for every path P from the root node to m, $h(m, P)$ never exceeds the cost (computed with reference to P) of the minimum cost path from m to a goal node.

(ii) In the definitions of *consistent* explicit graph and *psp*, replace $c(m, m_i)$ with $c(P, m, m_i)$.

(iii) Let $P'_1, P'_2, \ldots, P'_k$ be the paths in $G'_j(m)$ from m to tip nodes, and let P' be a path from s to m in $G'_j(m)$. Define

(a) $$Q'_j(m, P') = \min_{1 \leqslant i \leqslant k} \left[\max_{m' \in P'_i} \{c(P', P'_i, m') + h(m', P')\} \right].$$

Here $c(P', P'_i, m')$ is the cost of the path P'_i from m to m' as defined by path P'; $c(P', P'_i, m)$ is taken to be zero. Also let $Q'_j(n, P') = h(n, P')$ when n is a tip node. If there are no paths from m to a tip node and m itself is not a tip node then $Q'_j(m, P') = \infty$. By Q'_j we mean $Q'_j(s, \_)$.

(b) $$Q'_j(m) = \max(Q'_j(m, P'))$$

where the maximum is taken over all paths P' from s to m in the explicit graph G'_j.

(iv) Let $P_1, P_2, \ldots, P_{k'}$ be the paths in the search graph G from m to goal nodes, and let P be a path from s to m in G. Define

(a) $$Q(m, P) = \min_{1 \leqslant i \leqslant k'} \left[\max_{m' \in P_i} \{c(P, P_i, m') + h(m', P)\} \right],$$

where $c(P, P_i, m')$ indicates the cost of the path P_i from m to m'

defined by P. If n is a goal node let $Q(n, P) = 0$. If there are no solution paths passing through m let $Q(m, P) = \infty$. Thus $Q(m, P)$ is finite for every node m in G which lies on a solution path. By Q we mean $Q(s,\_)$.

(b) $$Q(m) = \max(Q(m, P)),$$

where the maximum is taken over all paths P from s to m in G.

With the above redefinitions, Lemma 3 remains true with appropriate changes in notation. It can be restated as follows:

Lemma 3. *Let m be a node with immediate successors $m_1, m_2, \ldots, m_k$ in G_j', and let P' be a path in G_j' from s to m. Then*

$$Q_j'(m, P') = \max_{1 \le i \le k} [h(m, P'), \min\{c(P', m, m_i) + Q_j'(m_i, P')\}].$$

Lemmas 4 and 6 remain the same as before. We impose, as in Section 2.1.1, a non-zero lower bound on the arc costs. GREC then terminates successfully, since Lemma 9 and Theorem 10 remain valid. When the arc costs and the heuristic estimates are path dependent, the b-value of a node can decrease with time; this cannot occur when the arc costs and heuristic estimates are constant. But $b(s)$ can never decrease with time even in the path dependent case and must finally achieve the value Q. Thus GREC, even with path dependent arc costs, outputs optimal solutions if heuristic estimates are admissible for every path P in the search graph.

3. One-machine job sequencing problems

3.1. Problem description

A one-machine minimum penalty job sequencing problem has the following general form. Jobs J_i with processing times $a_i > 0$, $1 \le i \le N$, 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* $G_i(\cdot)$, $1 \le i \le N$, 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 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

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

is minimized. The penalty functions are non-decreasing and in general nonlinear. Problems of this type have obvious relevance to industry, but are known to be hard to solve. In a more general setting, jobs can also 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_{ij}. 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_{ij} 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.* Existing approaches tend to make the assumption that setup times are sequence independent [13]. Solution methods that work in the absence of setup times can frequently be extended to the situation when setup times are sequence independent, since evaluation functions remain order preserving. But if we make the more realistic assumption that setup times are sequence dependent, evaluation functions become non-order-preserving and an A*-based graph search method can fail to output optimal solutions. Tree search procedures such as the branch-and-bound method remain applicable, but tend to be inefficient because many duplicate nodes get generated.

3.2. Classification of minimum penalty job sequencing problems

One-machine minimum penalty job sequencing problems can be classified into the following seven different types:

(a) Setup times are non-existent or sequence independent; evaluation functions in consequence are order-preserving:

(i) *Type* A: *Penalty functions are linear in job finish times.*

Problems of this type are simple. Ordering the jobs in non-decreasing order of a_k/p_k-values gives a minimum penalty sequence [6]; no searching is necessary.

(ii) *Type* B: *Penalty functions are quadratic in job finish times.*

Townsend [28] in his branch-and-bound formulation proposed the use of a novel evaluation function for the quadratic penalty case that turned out to be quite selective. Bagga and Kalra [2] and Gupta and Sen [7] subsequently made some improvements in Townsend's method. All these schemes employ tree search. Sen and Bagchi [22,24] have recently implemented a far more efficient A*-based graph search scheme that uses Townsend's evaluation function with Bagga and Kalra's suggested modifications. A similar implementation using GREC has also been achieved. See Section 3.3.

(iii) *Type* C: *Penalty functions are general polynomials or more complex functions (such as exponentials) in job finish times.*

Both tree and graph search methods are applicable to problems of these types. Unfortunately, good heuristic estimate functions are yet to be found. Schild and Fredman [20] give some examples of general penalty functions.

(b) Setup times are sequence dependent, evaluation functions in consequence are non-order-preserving:
   (i) *Type* D: *Penalty functions are linear in job finish times.*
   In this case the evaluation function, when formulated in the usual manner, is non-order preserving. The problem can be solved in a natural way using GREC [23]. An alternative way is to redefine the evaluation function to make it order preserving so that A* graph search remains applicable [15,24]. See Sections 3.4 and 4.1.
   (ii) *Type* E: *Penalty functions are quadratic in job finish times.*
   In this case, the penalty function being nonlinear, it does not appear possible to modify the non-order-preserving evaluation function and make it order-preserving. GREC again solves the problem in a natural way [23]. See Sections 3.4 and 4.2.
   (iii) *Type* F: *Penalty functions are general polynomials in job finish times.*
   The approach of Section 4.2 can be extended to penalty functions that are polynomials in job finish times. Good evaluation functions are yet to be found, however.
   (iv) *Type* G: *Penalty functions are more complex functions (such as exponentials or non-integer powers) of the job finish times.*
   In order to apply the method of Section 4.2, it may be necessary to approximate the penalty function with a finite polynomial. However, as for type F problems, no good heuristic estimate functions have been reported yet.

3.3. Tree search and graph search

Suppose jobs have no setup times. Consider the branch-and-bound (tree search) procedure of Townsend for the quadratic penalty job sequencing problem. A node in the tree corresponds to an ordered partial sequence of jobs. The node n_i represents the single job J_i, while the node n_{ij} represents the ordered two-job sequence (J_iJ_j), and so on. The root node corresponds to the null sequence. All nodes are therefore distinct. When a node corresponding to a partial sequence of k jobs gets expanded, $N - k$ sons are generated, each son being obtained by appending one of the remaining $N - k$ jobs to the partial sequence of k 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 is solved when a complete ordered sequence of N jobs gets selected from OPEN.

In the corresponding graph formulation, nodes 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 generated $N - k$ sons, each son being a subset of $k + 1$ jobs. As before, the cost of an edge is the penalty associated with the corresponding job. A problem gets solved when the complete set of N jobs gets selected from OPEN. The root is at level 0, and a node corresponding to a subset of k jobs is at level k. There are $C(N, k)$ nodes at level k, and each node has $k!$ incoming paths from the root. In contrast, a tree has

$N!/(N-k)!$ nodes at level k, and each node has only one incoming path. Thus there is a reduction in the number of nodes at level k by a factor $k!$. This reduction factor is actually an overestimate, since there is one other point to be taken into account. In graph search we need to keep in memory all the nodes generated up to a given time instant, including the expanded nodes, while in tree search we need not keep track of the expanded nodes. The total number of nodes in the graph at levels 0 through k equals $1+N+C(N,2)+\cdots+C(N,k)< kC(N,k)$ for $k \leqslant N/2$, which is less than the number of nodes at level k of the tree by a factor of $(k-1)!$. This may be viewed as a truer estimate of the reduction factor. The use of Townsend's method of computing the evaluation function yields a fast and memory efficient graph implementation of A* or GREC.

How much better is graph search compared to best-first or depth-first tree search? This has been checked out experimentally [22]. Graph search could be run upto 30 jobs, but best-first tree search ran only upto 16 jobs; beyond this point the available memory was exhausted. For 16 jobs, graph search was more than 40 times faster than best-first tree search. Depth-first search could in principle be run for a large number of jobs but was found to be markedly slower than graph search; for example, for 20 jobs it ran about 175 times slower.

When setup times are sequence dependent, the tree search method remains essentially the same, but the representation of nodes in graph search needs to be modified. Nodes can no longer correspond just to subsets of jobs, because the last job that has been processed must be remembered. 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. So 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. Thus the number of nodes at a level k in the search graph is 1 for $k=1$ or N, and $kC(N,k)$ for $0<k<N$. The reduction in the number of nodes over a tree representation is by a factor of $(k-1)!$ for $0<k<N$. Again, the true reduction factor is smaller and only about $(k-2)!$, since graph search must store all generated nodes in memory. In this case the evaluation function is non-order-preserving as explained below.

3.4. Non-order-preserving evaluation functions

Let us suppose setup times are sequence dependent. Let P_1 and P_2 be two paths from the root node s to *node n*. The jobs processed along the two paths all belong to the set 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

Table 1
Job setup and processing times

Job	Setup times				Processing times
	1	2	3	4	
1	–	1	1	∞	1
2	1	–	3	∞	4
3	∞	∞	–	10	3
4	∞	∞	∞	–	10

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 estimate are path dependent [1]. It is possible for a complete sequence of N jobs that is an extension of the subsequence determined by P_2 to have a lower total cost than a sequence of N jobs that is an extension of P_1. But a graph 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 reformulating the evaluation function in A*, but the strategy does not work when the penalties are nonlinear, in which case an algorithm like GREC must be used.

Example 17. Consider a linear penalty problem with 4 job where the penalty coefficients are 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 penalties 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 ith row and jth column of the table. The initial setup times s_{0j} are assumed to be 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 $(J_2 J_1 J_3 J_4)$.

4. Running GREC with non-order-preserving evaluation functions

4.1. Linear penalties

When penalties are linear but setup times are sequence dependent, the evaluation function when defined in the normal way is non-order-preserving. It is possible to make it order preserving by adjusting the evaluation function (see Appendix B). GREC can solve the problem in the same way, but there is the following more natural formulation which does not require any modification in the evaluation function. 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. There are many paths from the root to n, and at a subsequent instant the EXPLORE procedure can reach n again along another path; this time the completion time of job J_i is t_i' which is in general different from t_i. T is then reset to t_i'. The value of $b(n)$ depends on the path by which n is reached from the root, and thus it depends on T. When penalties are linear, $b(n)$ can be expressed as

$$b(n) = \alpha T + \beta ,$$

where α and β are parameters that depend on node n but not on the path taken to node n. In fact, if Q is the set of jobs remaining to be processed at n,

$$\alpha = \sum \{p_k \mid J_k \text{ is in } Q\} ,$$

and

$$\beta = b\text{-value of node } n \text{ taking node } n \text{ as the root}$$
$$\text{(or equivalently, taking } T = 0) .$$

Note that for any job J_k in Q, the finish time t_k can be expressed as $T + (t_k - T)$; the coefficients of T can be grouped together yielding the parameter α, and the remaining terms can be combined yielding β. The value of α remains constant during the search, but β changes as more and more jobs in Q get sequenced. When T changes in value from t_i to t_i', $b(n)$ changes in value by the amount $(t_i' - t_i)\alpha$ which does not depend on β. Thus when a node n is reached for the first time along a path P, GREC sets $b(n) = h(n, P)$, which is the heuristic estimate based on path P; GREC also computes α, and stores $b(n)$, α and T at the node. At a later instant, when n is reached again along a different path, the change in $b(n)$ is computed using α and the difference in T-values, and $b(n)$ is modified accordingly.

Example 18. Consider the job sequencing problem of Example 17. Let us assume for simplicity that all nodes have zero heuristic estimates. GREC outputs the minimum penalty sequence $(J_2J_1J_3J_4)$ with cost equal to 30 without expanding any node more than once. The algorithm operates as explained in Example 1. When the node $m = (\{1, 2, 3\}, 3)$ has been reached for the first time from the root at some instant along the path P_1 (i.e., when the job processing sequence is $J_1J_2J_3$), the values $T = 12$, $b(m) = 32$ and $\alpha = 1$ are stored at node m. When m is reached again from the root at a subsequent instant along path P_2 (the job sequence being $J_2J_1J_3$) with $T = 10$, the change in b-value at m is computed as $(10 - 12) \cdot \alpha = -2$. GREC next explores m with $b(m) = 30$ and outputs the optimal solution path.

4.1.1. Experiments

In the experiments, the heuristic estimate $h(n)$ at a tip node n was computed as follows [15]. 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 $Q' = Q \cup \{J_i\}$. 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 $(s_k + a_k)/p_k$-values, compute the finish times t_k of the jobs, and let the heuristic estimate be $\Sigma\, p_k t_k$, where the summation is over all jobs in Q. This yields a consistent heuristic (see Appendix B for proof). Our experimental results in Table 2 and Fig. 4 show that the heuristic estimate is quite selective.

A* tree search, GREC and depth-first branch and bound (DFBB) were coded in C and run on a UNIX-based DEC 5900 system. The 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 ranges 1 to 9 and 0 to 9 respectively, and were generated in the same way. The initial setup times s_{0j} 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. 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, nodes were stored in a node table, and successor lists were maintained for every expanded node. A hashing scheme was used for checking whether a newly generated successor was already present in the explicit graph. As in DFBB, the successors of a node were generated in non-decreasing order of effective job processing time. Since job sequencing problems have well-defined depth bounds, iterative deepening schemes like IDA* or its variants [11] run even slower than depth-first search, so we do not report running times for IDA*.

The results of our experiments are shown in Table 2. In Fig. 4, we plot the speedup ratios and node reduction ratios of algorithm GREC against the number of jobs. The ratios were computed with respect to A* tree search (plots labelled A) and DFBB (plots labelled B). We summarize our experimental observations below:

(i) The number of nodes generated and expanded both increased rapidly with the number of jobs for all the three algorithms. With 22 jobs, A* tree search

Table 2
Linear penalty functions

Number of jobs	GREC			A* (tree)			Depth-first		
	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded
12	0.02	363	70	0.04	793	103	0.06	1297	223
14	0.06	699	126	0.12	2000	220	0.21	3770	537
16	0.14	1471	252	0.44	5617	541	0.78	11 427	1388
18	0.35	2912	473	1.44	15 133	1272	2.66	31 567	3237
20	0.87	6082	954	4.59	41 154	3134	9.66	96 911	8990
22	2.55	12 172	1901						
24	6.65	21 823	3214						
26	18.46	40 633	6027						

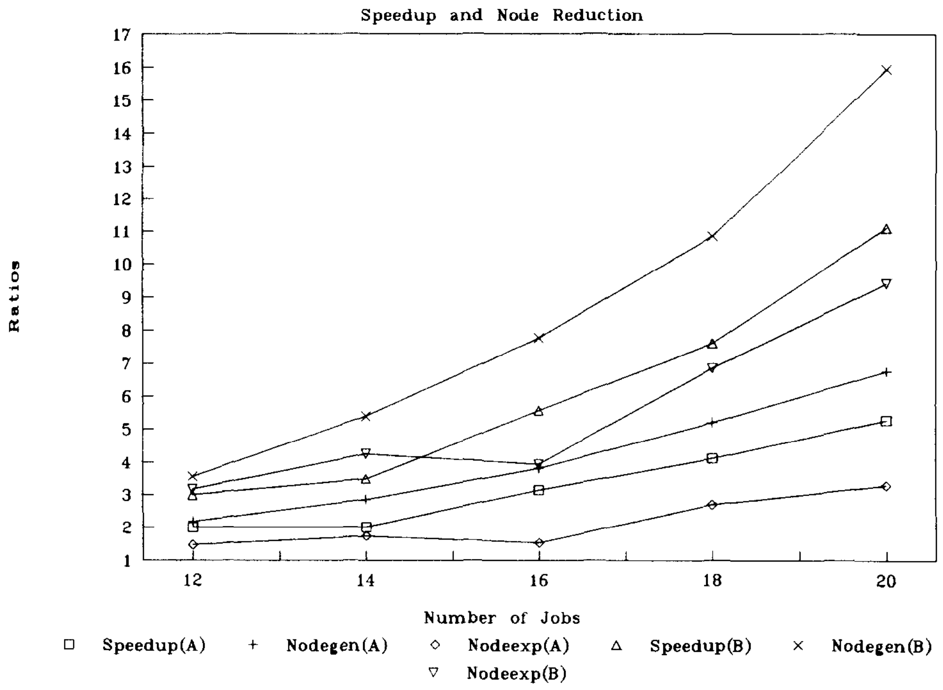

Fig. 4. Linear penalty functions.

sometimes ran short of memory, generating more than a million nodes in certain problem instances.

(ii) GREC was able to solve problem instances of larger size than A* tree search. We include results on GREC upto 26 jobs.

(iii) A variant of GREC was also implemented in which nodes of the explicit graph were stored but successor lists were not maintained. This variant ran slower than GREC because the time to generate successors and order them is high for job sequencing problems.

(iv) A* graph search using the modified evaluation function was also implemented. It was found that the numbers of nodes generated and expanded in A* graph search were close to the corresponding figures for GREC and the running time was comparable to that of GREC.

(v) DFBB is capable of solving problem instances of larger size because of its low memory needs [30]. Thus DFBB can be used to solve problem instances of much larger size than GREC. We ran it upto 20 jobs to compare its performance with GREC and A* tree search.

(vi) For 20 jobs, GREC ran around 5 times faster than A* tree search, and about 11 times faster than DFBB. The speedup ratios and node reduction ratios increased rapidly with the number of jobs. The speedup ratio was smaller than the node generation reduction ratio but higher than the node expansion reduction ratio. This could be the result of the combined effect of the following factors: the dependence of the branching factor of a node on its level in the graph, the consistency of the heuristic estimate, and the overhead of the algorithm.

(vii) DFBB needs very little memory and its overhead is low. But, even with successors generated in non-decreasing order of effective processing times, it ran slowly because it generated too many nodes.

4.2. *Quadratic penalties*

With quadratic penalty functions and no setup times, the evaluation function is order-preserving. Existing graph search algorithms are directly applicable. An efficient A* graph implementation can be realized using Townsend's method for computing the evaluation function. The problem can also be solved using GREC, and its performance is similar to that of A*. Unfortunately, it does not appear possible to generalize the A* graph implementation to sequence dependent setup times. The GREC implementation, on the other hand, can be readily extended to handle this case. The method of solution is similar to that described in Section 4.1; some additional computations are needed at each step as explained below because the penalty function is nonlinear.

The *b*-value at a node *n* in the UPDATE procedure cannot be computed in the same way as in the linear case. The *b*-value at a node *n* now has the form

$$b(n) = \alpha T^2 + \beta T + \gamma\ ,$$

where α, β and γ are parameters that depend on node *n* but not on *T*. Let *Q* be the set of jobs remaining to be processed at *n*. We have

$$\alpha = \sum \{p_k \mid J_k \text{ is in } Q\}\ ,$$

$$\beta = 2\sum \{p_k t_k \mid J_k \text{ is in } Q\}\ ,$$

and

$$\gamma = \text{the } b\text{-value at node } n \text{ with } n \text{ viewed as the origin}$$
$$\text{(i.e., with } T \text{ taken as zero)}\ .$$

As in the linear case, for any job J_k in *Q*, the finish time t_k can be expressed as $T + (t_k - T)$; the coefficients of T^2 can be grouped together yielding the parameter α, the coefficients of *T* yield the parameter β, and the remaining terms combine to give γ. The value of α remains constant during the search, β and γ change as more and more jobs in *Q* get sequenced; α, β and *T* must be stored at each node and reset in UPDATE. This increases the amount of computation and explains why GREC runs slower with quadratic penalties than with linear penalties.

This method can be extended to penalty functions that are higher powers of the job finish times. For example, in the cubic case $b(n)$ can be written as

$$b(n) = \alpha T^3 + \beta T^2 + \gamma T + \delta ,$$

where the parameters can be defined in a manner similar to that shown above. Extension to penalty functions that are polynomials in job finish times is immediate. Since good heuristic functions are not yet known, no experiments were performed with such penalty functions.

4.2.1. Experiments

At a tip node n, the effective processing time $s_k + a_k$ for a job J_k in Q is found as before and a heuristic estimate $h(n)$ is computed using a modification of Townsend's method. Townsend derived sufficient conditions for a sequence of N jobs to be minimum cost. He showed that a sequence is minimum cost if for every pair of adjacent jobs (J_i, J_j) where J_i precedes J_j, the following two conditions are both satisfied:

(i) $p_i/a_i \geq p_j/a_j$;
(ii) $p_i \geq p_j$.

The two conditions might not be satisfied simultaneously by an adjacent pair of jobs. Hence a search method is needed. To determine a lower bound at a node n in the branch-and-bound search, order the m jobs in Q as follows:

$$p_{i(1)}/a_{i(1)} \geq p_{i(2)}/a_{i(2)} \geq \cdots \geq p_{i(m)}/a_{i(m)} .$$

The penalty F' of the above subsequence of jobs is

$$F' = \sum \{p_{i(k)} t_{i(k)}^2 \mid 1 \leq k \leq m\} ,$$

where

$$t_{i(k)} = T + \sum \{a_{i(u)} \mid 1 \leq u \leq k\} ,$$

and

$$T = \sum \{a_k \mid J_k \text{ is in } S\} .$$

The lower bound can be obtained by considering all pairs of jobs (J_k, J_j) in the above ordered sequence where J_k precedes J_j and $p_k < p_j$, and for every such pair, reducing the penalty F' by $(p_j - p_k)a_j a_k$. In the presence of setup times, effective processing times must be used in place of actual processing times. The resulting heuristic estimate function is consistent (see Appendix B for proof). The experimental data indicated that the heuristic estimate was less selective in this case than in the absence of setup times.

In the experiments, the programs were written in C and run on a UNIX-based DEC 5900 system as before. Processing times of jobs were integers and were chosen randomly in the range 1 to 99 from a uniform distribution. Penalty coefficients were integers in the ranges 1 to 9 and were generated in the same

Table 3
Quadratic penalty functions—Set I

Number of jobs	GREC			A* (tree)			Depth-first		
	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded
12	0.09	920	232	0.15	3685	488	0.20	5216	789
14	0.28	2287	539	0.69	12 961	1446	0.90	18 853	2342
16	0.83	5544	1277	3.39	50 776	5003	4.75	80 575	8662
18	3.25	14 526	3322						
20	13.25	36 646	8193						

way. For both sets the initial setup times s_{0j} were assumed zero for simplicity. Two sets of setup times were considered: Set I consisted of setup times in the range 0 to 9, while Set II consisted of setup times in the range 1 to 5. The heuristic estimate for Set I was less selective and this increased the running time. GREC could be run for a larger number of jobs in a given time when Set II was used. 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. In implementing depth-first search, the successors of a node were generated in

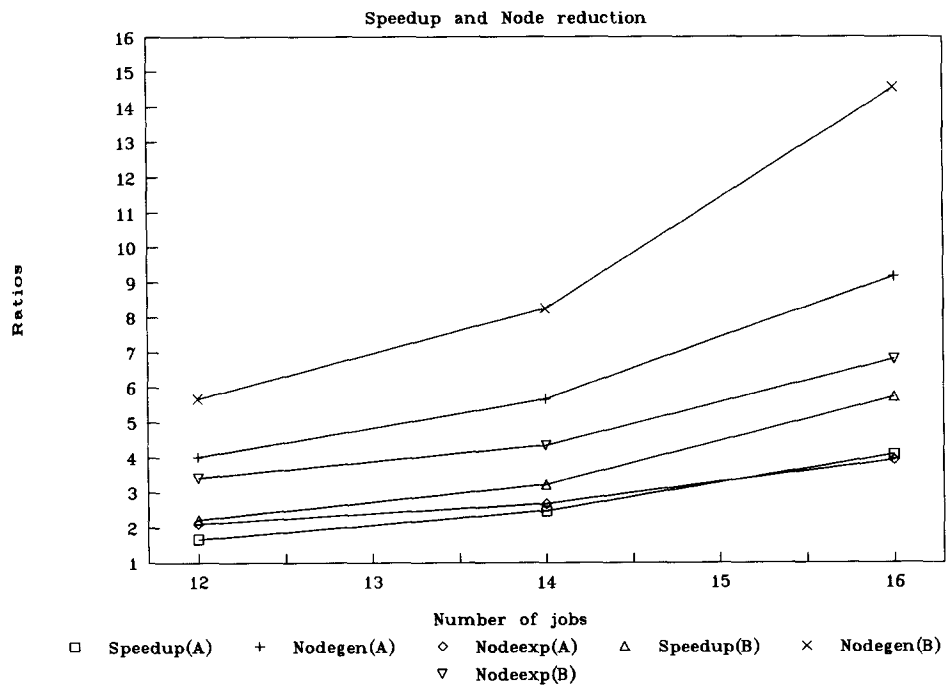

Fig. 5. Quadratic penalty functions: Set I.

Table 4
Quadratic penalty functions—Set II

Number of jobs	GREC			A* (tree)			Depth-first		
	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded	Time (secs)	Nodes generated	Nodes expanded
12	0.05	649	163	0.12	2906	388	0.15	3937	594
14	0.16	1482	349	0.56	10 344	1169	0.69	14 499	1805
16	0.55	3420	804	3.23	47 120	4750	3.91	66 756	7214
18	1.60	7517	1687				16.92	233 250	21 689
20	4.92	18 158	4052				85.03	979 937	82 548
22	30.64	40 555	9219						
24	91.09	81 714	17 659						

non-decreasing order of a_i/p_i' where p_i' refers to the effective processing time of job J_i.

Experimental results for Set I are shown in Table 3 and Fig. 5, and for Set II in Table 4 and Fig. 6. The results were similar in trend to those for the linear case. GREC could be run for a larger number of jobs, and was about five times faster than A* tree search for 16 jobs. For 18 jobs, tree search generated more than one million nodes in some instances. Since the heuristic estimate is not as sharp as in

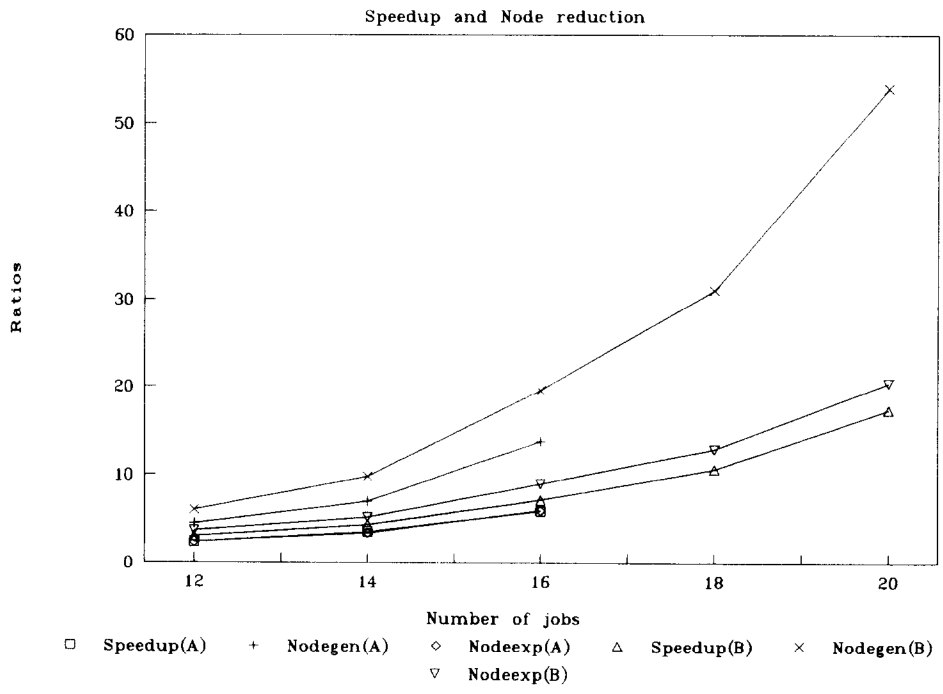

Fig. 6. Quadratic penalty functions: Set II.

the linear case, running times in Tables 3 and 4 are larger than in Table 2. The graphs in Figs. 5 and 6 showed an increase of speedup and node reduction ratios with the number of jobs. DFBB ran about six times slower than GREC for 16 jobs; as before, DFBB is capable of solving problems of much larger size than GREC.

5. Conclusion

Although A* is capable of graph search and not just tree search, in practice it has been generally used as a tree search procedure. It has been found recently that in some application areas such as one-machine job sequencing, graph search with A* can be much faster than best-first or depth-first tree search for problems of small and medium size. However, A* is not suitable for searching graphs when the evaluation function is non-order-preserving. In such cases the graph search algorithm GREC can be used in place of A*. Experimental results indicate that when setup times are sequence dependent, and the evaluation function is in consequence non-order-preserving, GREC solves linear penalty and quadratic penalty job sequencing problems faster than tree search schemes.

The detection and elimination of duplicate nodes is a most pressing issue in tree search algorithms. Taylor and Korf [27] suggest a technique for pruning duplicate nodes in depth-first search. Their method requires a preprocessing of the search graph by means of an exploratory breadth-first search. This determines the shortest sequence of operators that are needed to move from one node to another, and thus helps in detecting and eliminating duplicate nodes during the actual search phase. Preprocessing is needed only once for each type of problem, and the knowledge acquired can be used later during the depth-first search on the various problem instances. The implicit assumption here is that the cost of an arc between a pair of nodes is always the same and is independent of the problem instance. This is true for problems like the 15-puzzle, 24-puzzle and Rubik's cube. But for job sequencing problems, it is not possible to apply such techniques effectively since arc costs are not the same for different problem instances.

Very recently, Reinefeld and Marsland [17] have reported some enhancements of iterative deepening search. They have shown that search efficiency can be greatly improved by storing, in a table in memory, nodes which have already been encountered in the course of the search. Their search algorithm TRANS appears to perform much better than IDA* on both the 15-puzzle problem and the graph-based solution method suggested by Pearl [14] for the travelling salesman problem. TRANS is conceptually very close to GREC, as the pseudo-code given in [17, Fig. 7, p. 709] shows. The hash table used in TRANS to store generated nodes is similar to the table used in GREC. However in TRANS, when the hash table becomes full, existing entries can be overwritten by more promising new entries. Another difference between the two algorithms is that TRANS only stores the nodes of the explicit graph but not the successor lists. This can be an advantage in problems like the 15-puzzle where node generation time is in-significant. But it is a disadvantage in job sequencing, where it takes time to generate nodes; since the same node can be explored many times, higher speeds can be achieved if successor lists are maintained in memory. Even in tree search problems like the rectangular cutting stock problem [21,29], it is helpful to know the node numbers of successors since nodes can be repeatedly explored and node generation time is high; in such cases every node is distinct and hashing serves no useful purpose. In fact, in this case a separate array for successors is not needed; it is enough to store with each expanded node the node number of the first son and the number of sons, since the sons can be assigned successive node numbers at the time of generation.

In ending, we point out that GREC has one serious limitation. Since it stores the entire explicit graph in memory it cannot solve large problems. Is there any way to get around this difficulty? At this time we do not know the answer. In job sequencing problems, the use of an iterative deepening scheme is inadvisable; the total number of jobs being known, there is a well-defined depth bound, so depth-first search runs faster than iterative deepening schemes. For the same reason, MREC as currently formulated is unsuitable if the available memory M is small. Algorithms SMA* [18] and MA* [5] are based on A* and their graph implementations will run into the same difficulties as A* graph search. An efficient graph-based memory constraint algorithm that maintains the entire explicit graph in memory appears to be the need of the hour. Is it possible to improve the memory management features of MREC so that it becomes capable of solving sequencing problems efficiently without storing the entire explicit graph? Alternatively, can successor lists be incorporated in TRANS and the memory management technique modified so that it runs efficiently on sequencing problems of larger size?

Appendix A. Properties of GREC: proofs of claims in Section 2.1

Lemma 6. *Let j be any instant during the execution of GREC. Then*
(i) $b_j(m) \leq Q'_j(m)$ *for any node m in G'_j;*
(ii) $b_j \leq Q'_j \leq Q$.

Proof. For a tip node, (i) is true by definition. For a non-tip node, we prove the result by contradiction. Let j be the earliest instant at which the lemma fails. The lemma must fail at a node at which recalculation takes place at instant j. This is so because if m is a non-tip node at which recalculation does not take place at instant j, then m is a non-tip node at instant $j-1$ and by assumption and $b_{j-1}(m) \leq Q'_{j-1}(m)$; we know by Lemma 4 that $Q'_{j-1}(m) \leq Q'_j(m)$, so $b_j(m) = b_{j-1}(m) \leq Q'_j(m)$. In the course of recalculation at instant j, let n be the first node (in time) at which the lemma fails. Let $n_1, n_2, \ldots, n_k$ be the immediate successors of n. At the moment the recalculation takes place at n, the lemma holds for each of its immediate successors. Some of these successors will have had their b-values updated already this instant, some others will not change their b-values at all, and in case n lies in a loop, there may be an immediate successor of n whose b-value will be updated after n. However, when the recalculation is done at n, for $1 \leqslant i \leqslant k$,

$$b(n_i) = h(n_i) = Q'_j(n_i) \quad \text{if } n_i \text{ is a tip node}$$

and

$$b(n_i) \leqslant Q'_j(n_i) \quad \text{if } n_i \text{ is a non-tip node}$$

so that

$$\begin{aligned} b_j(n) &\leqslant \max_{1 \leqslant i \leqslant k} \{b_{j-1}(n), \min[c(n, n_i) + b(n_i)]\} \\ &\leqslant \max_{1 \leqslant i \leqslant k} \{Q'_{j-1}(n), \min[c(n, n_i) + Q'_j(n_i)]\} \\ &\leqslant Q'_j(n) \quad \text{by Lemmas 3 and 4}. \end{aligned}$$

This is a contradiction.

When the heuristic estimate function is admissible the b-value at m can never exceed $Q(m)$ even at termination, since it can never be set to a value exceeding $Q(m)$ from above. If the heuristic is inadmissible, then at termination there might be a node m on the output path, where m is not the root, whose b-value is greater than $Q(m)$ because it has been set to a large value from above. If GREC does not terminate, the b-value as well as the Q'-value at m would get revised upwards from below and at the next instant j we would again have $b_j(m) \leqslant Q'_j(m)$. The b-value at the root cannot be set from above, so whether the heuristic is admissible or inadmissible, $b_j \leqslant Q'_j$ at every instant j during the execution of GREC; a similar inequality would also hold at termination. □

Lemma 9. *Let j be any positive integer. If GREC has not terminated by instant j then at some instant $j' \geqslant j$ prior to termination, $G'_{j'}$ is consistent.*

Proof. Suppose the lemma is false. Then at every instant $j' > j$, GREC is still running and the explicit graph $G'_{j'}$ is not consistent. Since no tip node gets explored, no nodes get expanded and $G'_{j'}$ remains unchanged, but b-values of non-tip nodes which do get explored increase by at least τ. Thus at every instant $j' > j$, $b_{j'}$ increases by at least τ. Since $G'_{j'}$ remains the same, $Q'_{j'} = Q'_j$ for every instant $j' > j$. So, as Q'_j is finite, a time must come when $b_{j'} > Q'_{j'}$, which contradicts Lemma 6. □

Theorem 10. *Algorithm GREC terminates successfully*, i.e., *it finds a goal node and outputs a solution path.*

Proof. By Lemma 9, GREC cannot go on running indefinitely without expanding a tip node. If j and j' are two instants at which tip nodes get expanded, then the psps at these two instants must be distinct. If the search graph is finite and loop free then GREC obviously terminates. If not, we recall that the cost of an arc is at least $\tau > 0$, a node has finitely many immediate successors, and $b(s)$ cannot exceed Q; it follows that GREC must terminate in this case too. □

Definition 11. Let b_{GREC} be the b-value at the root s when GREC terminates.

Theorem 12.
(i) $b_{\text{GREC}} = Q$.
(ii) $b_{\text{GREC}} = b^*$ *if the heuristic estimate function is admissible.*

Proof. (i) By Lemma 6(ii), $b_{\text{GREC}} \leqslant Q$. But since a solution path is found at termination, $b_{\text{GREC}} \geqslant Q$.
(ii) If the heuristic is admissible, $Q = b^*$. □

Theorem 14. *All nodes expanded by GREC belong to the set V.*

Proof. Immediate by Lemma 6(ii). □

Appendix B

B.1. Linear penalties: making the evaluation function order-preserving

Let node $m = (S, J_i)$ be an immediate predecessor of node $n = (S \cup \{J_k\}, J_k)$ in the explicit graph. Then, in the standard A* formulation, the cost of the currently known best path from the root to n is

$$g(n) = \min\{g(m) + p_k(T + s_{ik} + a_k)\}$$

where T is the completion time of m's last job J_i as determined by the currently known best path to m, and the minimum is taken over all immediate predecessors m of n in the explicit graph. This cost function is non-order-preserving, since the arc cost $c(m, n) = p_k(T + s_{ik} + a_k)$ depends on the path by which m is reached from the root. We get over the problem in the following way. Let P be a solution path, and let jobs be sequenced in the order $J_1, J_2, \ldots, J_N$ along P. Then the cost of P is

$$\begin{aligned} &p_1a_1 + p_2(a_1 + s_{12} + a_2) + \cdots + p_N(a_1 + s_{12} + \cdots + a_N) \\ &\quad = a_1(p_1 + p_2 + \cdots + p_N) + (s_{12} + a_2)(p_2 + \cdots + p_N) \\ &\quad\quad + \cdots + (s_{N-1,N} + a_N)p_N . \end{aligned}$$

Taking a cue from this expression, we define new g-values which we call *deferral* g-values as follows:

$$g_{\mathrm{d}}(n) = \min\Big\{ g_{\mathrm{d}}(m) + (s_{ik} + a_k) \sum p_j \Big\} ,$$

where the minimum is again taken over all predecessors m of n, and the summation is over all jobs remaining to be processed at n. The g_d-values are no longer path dependent and the evaluation function is order-preserving, so A*-based graph search can still be employed [15,24]. The heuristic estimate at a node now estimates the contribution to the total penalty of the jobs remaining to be processed; the total contribution of the jobs already processed has been taken into account in the expression for the g_d-value.

B.2. Linear penalties: consistency of the heuristic estimate $h(\cdot)$

We show that the heuristic estimate $h(n, P)$ in the linear penalty case (Section 4.1.1) is consistent; the path P is shown as a parameter since the heuristic estimate depends on the finish time of the last job at node n.

Let node m be a parent of node n. We have to show that

$$h(m, P) \leqslant c(P, m, n) + h(n, P)$$

where $c(P, m, n)$ is the cost of the arc between m and n along path P; $c(P, m, n)$ depends on the actual time taken to process the last job J_i at node n and also on the path from s to n. The jobs remaining to be processed at node m all belong to $Q \cup \{J_i\}$. Let J be such a job, $J \neq J_i$. Then the effective processing time of J at m is no larger than its effective processing time at n. Moreover, the actual time taken to process J_i, which determines $c(P, m, n)$ is not smaller than its effective processing time at node m. Thus the left-hand side of the above expression can be no larger than the right-hand side.

B.3. Quadratic penalties: consistency of the heuristic estimate $h(\cdot)$

The method of computation of the heuristic estimate is described in Section 4.2.1. The method ensures that the heuristic estimate function is admissible. We argue that it is also consistent. Let node n be a son of node m, and let P be a path from s to m. We want to show that

$$h(m, P) \leqslant c(P, m, n) + h(n, P) .$$

Let the edge (m, n) correspond to job J. When the heuristic is computed at m, the jobs remaining to be processed are sequenced in non-decreasing order of a_i/p_i values. Job J occurs somewhere in this sequence. To derive the heuristic $h(m, P)$, the penalty for this sequence is computed, and then correction terms corresponding to certain pairs of jobs are deducted. Thus we can say that the left- and right-hand sides in the above inequality differ in the following respects:

(i) In the right-hand side, J occurs as the first job, while in the left-hand side it is not necessarily the first job.

(ii) In the right-hand side, correction terms have not been deducted from the penalty for any job pair of which J is a member, but such corrections have been made in the left-hand side.

Consider an optimal path P from m to a goal. If J is the first job along P, then the above inequality must hold, since the computation procedure for the heuristic ensures that the contribution of J to $h(m, P)$ is only a lower bound on the final contribution of J to the total penalty. If J is not the first job along P, then forcing J to the front and not making any corrections for J can only increase the value of the right-hand side.

References

[1] A. Bagchi and A. Mahanti, Search algorithms under different kinds of heuristics: a comparative study, *J. ACM* **30** (1983) 1-21.

[2] P.C. Bagga and K.R. Kalra, A node elimination procedure for Townsend's algorithm for solving the single machine quadratic penalty function scheduling problem, *Manage. Sci.* **26** (1980) 633-636.

[3] C.E. Bell and K. Park, Solving resource-constrained project scheduling problems by A* search, *Nav. Res. Logist.* **37**, (1990) 61-84.

[4] L. Bianco, S. Ricciardelli, G. Rinaldi and A. Sassano, Scheduling tasks with sequence dependent processing times, *Nav. Res. Logist.* **35** (1988) 177-184.

[5] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. De Sarkar, Heuristic search in restricted memory, *Artif. Intell.* **41** (1989) 197-221.

[6] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop* (Ellis Horwood, Chichester, 1982).

[7] S.K. Gupta and T. Sen, On the single machine scheduling problem with quadratic penalty function of completion times: an improved branching procedure, *Manage. Sci.* **30** (1984) 644-647.

[8] P.E. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum-cost paths, *IEEE Trans. Syst. Sci. Cybern.* 4 (2) (1968) 100-107.

[9] H. Kaindl and A. Khorsand, Memory-bounded bidirectional search, in: *Proceedings AAAI-94*, Seattle, WA (1994) 1359-1364.

[10] R.E. Korf, Depth-first iterative deepening: an optimal admissible search, *Artif. Intell.* **27** (1985) 97-109.

[11] R.E. Korf, Linear-space best-first search: summary of results, in: *Proceedings AAAI-92*, San Jose, CA (1992).

[12] J.D.C. Little, K.G. Murty, D.W. Sweeny and G. Karel, An algorithm for the travelling salesman problem, *Oper. Res.* **11** (1963) 972-989.

[13] A.J. Mason and E.J. Anderson, Minimizing flow time on a single machine with job classes and setup times, *Nav. Res. Logist.* **38** (1991) 333-350.

[14] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).

[15] R. Ramaswamy and A.K. Sen, Single machine scheduling as a graph search problem with path-dependent arc costs, in: *Proceedings ECAI-92*, Vienna (1992) 11-15.

[16] C.R. Reeves, ed., *Modern Heuristic Techniques for Combinatorial Problems* (Blackwell Scientific Publications, Oxford, 1993).

[17] A. Reinefeld and T. Marsland, Enhanced iterative-deepening search, *IEEE Trans. Pattern Anal. Mach. Intell.* **16** (1994) 701-709.

[18] S. Russell, Efficient memory-bounded search methods, in: *Proceedings ECAI-92*, Vienna (1992) 1-5.

[19] S. Sahni and T. Gonzalez, p-complete approximation problems, *J. ACM* **23** (1976) 555-565.

[20] A. Schild and I.J. Fredman, Scheduling tasks with deadlines and non-linear loss functions, *Manage. Sci.* **9** (1963) 73-81.

[21] A.K. Sen and A. Bagchi, Fast recursive formulations for best-first search that allow controlled use of memory, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 297-302.

[22] A.K. Sen and A. Bagchi, Job sequencing with quadratic penalties: an A*-based graph search approach, in: *Proceedings IEEE Conference on Artificial Intelligence for Applications*, Orlando, FL (1993) 190–196.

[23] A.K. Sen and A. Bagchi, Non-order-preserving evaluation functions: recursive graph-search methods for job sequencing problems, in: *Proceedings IJCAI-93*, Chambery (1993) 1423–1429.

[24] A.K. Sen, A. Bagchi and R. Ramaswamy, Searching graphs with A*: applications to job sequencing, *IEEE Trans. Syst., Man Cybern. Part A Syst. Humans* **26** (1996) 168–173.

[25] A.K. Sen, A. Bagchi and B.K. Sinha, Admissible search methods for minimum penalty sequencing of jobs with setup times on one and two machines, in: *Proceedings IJCAI-91*, Sydney, NSW (1991) 178–183.

[26] E. Taillard, Some efficient heuristic methods for the flow shop sequencing problem, *Eur. J. Oper. Res.* **47** (1990) 65–74.

[27] A. Taylor and R.E. Korf, Pruning duplicate nodes in depth-first search, in: *Proceedings AAAI-93*, Washington DC (1993).

[28] W. Townsend, The single machine problem with quadratic penalty function of completion times: a branch and bound solution, *Manage. Sci.* **24** (1978) 530–534.

[29] K.V. Viswanathan and A. Bagchi, Best-first search methods for constrained two-dimensional cutting stock problems, *Oper. Res.* **41** (1993) 768–776.

[30] W. Zhang and R.E. Korf, Depth-first vs. best-first search: new results, in: *Proceedings AAAI-93*, Washington DC (1993).