Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory

Anup K. Sen and A. Bagchi
Computer Aided Management Centre
Indian Institute of Management Calcutta
P. O. Box 16757, Calcutta - 700 027, INDIA

Abstract

MREC is a new recursive best-first search algorithm which combines the good features of A* and IDA*. It is closer in operation to IDA*, and does not use an OPEN list. In order to execute, all MREC needs is sufficient memory for its implicit stack. But it can also be fed at runtime a parameter M which tells it how much additional memory is available for use. In this extra memory, MREC stores as much as possible of the explicit graph. When M = 0, MREC is identical to IDA*. But when M > 0, it can make far fewer node expansions than IDA*. This can be advantageous for problems where the time to expand a node is significant. Extensive runs on a variety of search problems, involving search graphs that may or may not be trees, indicate that MREC with M = 0 is as good as IDA* on problems such as the 15-puzzle for which IDA* is suitable, while MREC with large M is as fast as A* on problems for which node expansion time is not negligible.

1 Introduction

Among best-first search algorithms, A* is the most widely used, because it is relatively simple to implement and executes quite fast on the average. But it runs short of memory when trying to solve random instances of the 15-puzzle. IDA* [Korf, 1985], on the other hand, solves 15-puzzle problem instances with ease. It is also straightforward to implement, and unlike A*, uses very little memory. But it is not able to take advantage of any additional memory that may be available. In this paper we present a new recursive best-first search algorithm MREC that combines the good features of A* and IDA*. It is closer in operation to IDA* and does not use an OPEN list. It runs on both trees and graphs, and takes care of loops in a simple and natural way. To solve a problem, all MREC needs is enough memory for its implicit stack. But MREC can also be fed at runtime a parameter M indicating how much extra memory it can use. When M = 0, MREC and IDA* are essentially identical in operation. When M is large, MREC keeps the entire explicit search graph in memory in an effort to restrict the total number of node expansions. In fact, no node is expanded more than once by MREC when sufficient memory is available. This is advantageous for problems where node expansion is time consuming. At intermediate values of M, MREC keeps as much of the explicit search graph in memory as possible. In such cases, MREC may expand a node more than once. In computer experiments undertaken by us on a wide variety of search problems, MREC compares most favourably in running time with both A* and IDA*. MREC has the additional advantage that it readily generalizes to AND/OR graphs, and is even able to accommodate directed cycles in such graphs [Sen, 1988].

2 How MREC Operates

MREC can be viewed as a generalization of IDA*. The heart of the algorithm is a recursive procedure EXPLORE, which gets called upon at each iteration to explore the explicit search graph below the root node s. 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. If sufficient memory is available, it expands the tip node n and adds the new nodes and edges to the explicit graph; otherwise, it peeps below n in the manner of IDA*, performing what may be called a *virtual expansion* of n. (In this case, the successors of n and the corresponding edges do not get added to the explicit graph). As in IDA*, a cutoff value is used for monitoring the downward movement. Thus MREC has a very simple structure. For convenience of explanation we have broken up EXPLORE into two smaller procedures EXPAND and UPDATE. Each explored node n in the graph has an associated value b(n) which gives the best estimate currently known of the minimum cost of a path from n to a goal node; b(n) initially equals the heuristic estimate h(n). The procedure EXPAND expands a tip node and adds the newly generated nodes and edges to the explicit graph. The procedure UPDATE explores the graph below a node and updates the b-values of nodes. When memory is in short supply, it makes a virtual expansion of the tip node. In the algorithm below, we store the output solution path in *outpath*. The explicit search graph and its associated parameters are assumed to be accessible to all the procedures.

```
program MREC;
var terminate : boolean;
begin (* the root node s is in explicit graph *)
  terminate := false; b(s) := h(s);
  repeat EXPLORE(s,b(s)) until terminate;
  output b(s) as solution cost;
  output outpath as solution path;
end.

procedure EXPLORE(n:node;var bnode:integer);
begin
  if n is a goal node then
    begin terminate := true; return; end;
  if (n is a tip node) and (sufficient memory
      available for storing successors of n)
  then EXPAND(n);
  UPDATE(n,bnode);
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
        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 bnode:integer);
var cutoff,newb : integer;
begin
  cutoff := ∞;
  for each successor n_i of n (* n_i may or may not
    be present in the explicit graph *) do
      begin
        if n_i is in the explicit graph then
          newb := b(n_i)
        else newb := h(n_i);
        if newb + c(n,n_i) ≤ bnode then
          begin
            newb := bnode - c(n,n_i);
            EXPLORE(n_i,newb);
            if terminate then
              begin add n_i to outpath; return; end;
          end;
        if newb + c(n,n_i) < cutoff then
            cutoff := newb + c(n,n_i);
      end;
  bnode := cutoff;
  if n is in explicit graph then b(n) := bnode;
end;
```

Note that if for a successor n_i of n, we have $b(n_i) \leq b(n) - c(n,n_i)$, then n_i gets explored. If, however, we have $b(n_i) > b(n) - c(n,n_i)$ for every successor n_i of n, then b(n) gets updated to $\min\{b(n_i) + c(n,n_i)\}$. When memory is available, the explicit graph grows in size with more nodes and arcs getting added to it. When memory is not available, the explicit graph does not change and tip nodes remain tip nodes, but b-values of tip nodes increase as the exploration goes down deeper into the implicit search graph. When M is small, many nodes get expanded again and again, as in IDA*. With increase of M, the total time spent on node expansions goes down, but some time is consumed in updating the explicit graph.

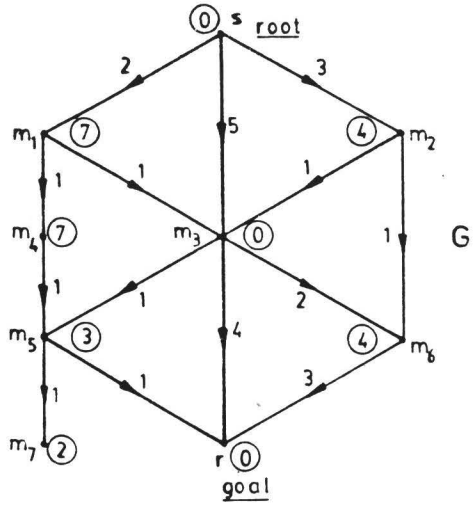

Fig. 1

Example: For the network G shown in Fig.1 the node expansion sequence when M = 0 is

s						
s	m_3					
s	m_3	m_2	m_3			
s	m_3	m_2	m_3	m_5	m_7	r

which is the same as that for IDA*. This assumes that successors are generated from left to right. With unlimited memory, the explicit graph at termination is shown in Fig.2. When the available memory can accom-

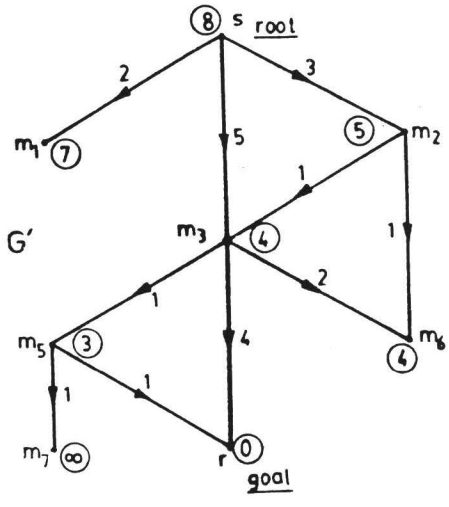

Fig. 2

modate two expanded nodes with their successors, the explicit graph at termination is shown in Fig.3. Solid lines indicate edges present in the explicit graph, while dotted lines indicate edges which have been explored but not saved in memory. Nodes not saved in memory are marked by '*'.

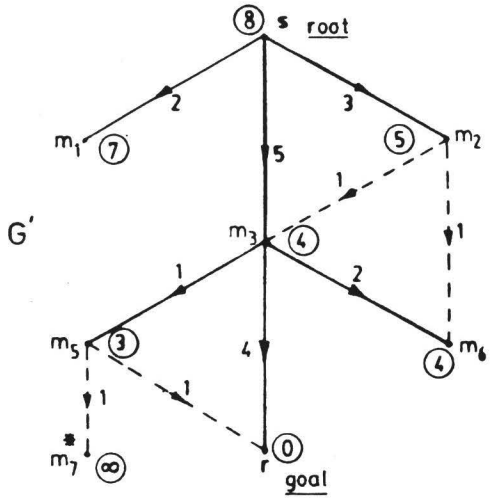

Fig. 3

We now enumerate some interesting properties of MREC:

1. When M = 0, MREC expands nodes in the same sequence as IDA*. For M > 0, the node expansion sequences would not in general be identical. However, suppose we ignore reexpansions of a node, and only consider expansions of new nodes by the two algorithms. Then the node expansion sequences would again be identical.

2. The node expansion sequence of MREC, even with unlimited memory, may not conform to that of A*. In general, MREC and A* output different solution paths.

3. MREC never expands a node more than once if sufficient memory is available for storing the explicit graph. When available memory is limited, MREC may expand a node repeatedly; but the total number of node expansions never exceeds the number made by IDA*.

4. When the heuristic estimate function is admissible, MREC finds a minimal cost solution path, just like A*. But the solution paths found by the two algorithms can be different. For inadmissible heuristics, even the costs of the output paths found by MREC and A* can be different. On the other hand, MREC and IDA* always output the same solution path.

5. The worst-case running time of MREC, like that for A* and IDA*, can be exponential in the number of nodes in the search graph.

3 Experimental Results

To compare the running times of MREC, A* and IDA*, we ran the three algorithms on a variety of search problems. Programs were written in PASCAL, and the

TABLE 1 : 8-puzzle

Mean Cost = 22.10 **No. of Runs = 100**

algorithm	time in secs		node gen		node exp	
	mean	std	mean	std	mean	std
A*	0.19	0.29	2898.65	3956.97	1737.09	2373.09
MREC (M=∞)	0.03	0.05	2710.13	3853.12	1628.47	2320.76
IDA*	0.04	0.06	3856.39	5538.16	2324.29	3338.72

TABLE 2 : 15-puzzle

Mean Cost = 53.72 **No. of Runs = 25**

algorithm	no. of nodes	time in secs		node gen		node exp	
		mean	std	mean	std	mean	std
IDA*		1242.59	1369.24	122927208	133519920	62436248	92273152
MREC	0	1063.93	1161.31	122927208	133519920	62436248	92273152
	50000	1099.05	1217.06	122742600	123490192	62341376	92199904
	200000	1057.23	1154.97	122334712	133392120	62133064	68508696
	600000	1089.59	1191.33	121476098	133143136	61695380	91641592

machine used was the VAX 8550. A large number of runs were taken for each of the following problems, and also for some other problems not shown here [Sen, 1988]. The tables in this paper give a selection of the results that were obtained. Care was taken to implement the algorithms as efficiently as possible. The OPEN set of A* was implemented as a priority queue. For search graphs that are not trees, a hashing technique was employed to check for duplicate nodes. In the tables, MREC (M = ∞) refers to the implementation of MREC assuming available memory is unlimited; it is then unnecessary to check in EXPLORE whether the memory bound has been exceeded. Again, when M = 0, EXPAND has no role to play since all expansions are virtual expansions; in this case, all references to EXPAND can be eliminated from the code.

a) **8-puzzle** and **15-puzzle** : The goal node was fixed, and random, solvable initial configurations were generated. The search graph was represented as a tree. The Manhattan distance function was used as the heuristic. This heuristic is known to be admissible and consistent. One hundred instances of the 8-puzzle were solved by A*, IDA* and MREC (Table 1). The running time of MREC does not vary with change of M, so we arbitrarily chose M to be large. The table shows the running times, the number of nodes generated, and the number of nodes expanded (mean and standard deviation). For the 15-puzzle, 25 randomly generated instances were solved using IDA* and MREC (Table 2). For MREC, M is specified in terms of the number of nodes that can be stored in memory. As can be seen, there is very little variation in the running time of MREC as M is changed. The reason is that the time to expand a node and to calculate the heuristic estimates of successors is small; it is of the same order as the time taken to update the explicit graph and to retrieve stored values from it.

b) **Travelling Salesman Problem (TSP_T)** [Little *et al.*, 1963] : The well known method of Little et al. for solving the travelling salesman problem employs a depth-first branch-and-bound technique. The search graph is a tree. The method can be easily modified to make the search best-first instead of depth-first. For the 30-city problem, 100 cost matrices were randomly generated. The matrices were asymmetric, and the costs did not necessarily satisfy the triangle inequality. As can be seen from Table 3, A* and MREC (M = ∞) take almost the same time. Since the expansion time of a node is appreciable, there is no point in running either IDA*, or MREC with small values of M, on this problem. The original depth-first formulation of [Little *et al.*, 1963] runs almost as fast as A* or MREC (M = ∞), and needs very little memory; on those grounds it would seem to be the preferred method.

TABLE 3 : TSP_T

Cities = 30 **Mean Cost = 1616.03**
No. of runs = 100

algorithm	**time in seconds mean**	**node gen mean**	**node exp mean**
A*	3.69	1342.64	670.82
MREC (M=∞)	3.57	1335.08	667.04

c) **Rectangular Cutting Stock Problem (CRGKP)** [Viswanathan and Bagchi, 1988] : Here we are given a single rectangular stock sheet S of length L and width W which must be cut (using guillotine cuts only) into N demanded rectangles of specified dimensions in such a way that total value is maximized *and* demand constraints on demanded rectangles are not violated. For given values of L, W and N, 25 problems were randomly generated, and A* and MREC (M = ∞) were run on them. The running times were almost the same for the two algorithms, since the node expansion time is very high (Table 4). The method of solution given in [Viswanathan and Bagchi, 1988] is such that depth-first methods become inapplicable, so IDA*, or MREC with small values of M, cannot be run at all.

TABLE 4 : CRGKP

Length = 50 **Width = 35** **No. of rectangles = 10**

Mean Optimal Value = 4974.40 **No. of runs = 25**

algorithm	**time in seconds mean**	**node gen mean**	**node exp mean**
A*	2.58	2425.04	286.80
MREC (M=∞)	2.55	2415.16	285.84

d) **General d-ary tree** : Suppose we are given a uniform d-ary tree of unlimited depth with bi-directional edges of unit cost. There is a single goal node at a distance N from the root. The heuristic estimate of a node n is given by h(n) = r.h*(n), where h*(n) is the actual length of the shortest path from n to a goal node, and r is a random number between 0 and 1. The problem is to find a minimal cost solution path using a best-first search method (assuming of course that N is not known to the method). For d = 9 and N = 6, 100 problem instances were generated. The goal node position (at the specified depth) was randomly selected and h*-values were computed for all nodes; h-values were then generated using random values of r and tabulated in advance. These h-values were used in the search, and results are shown in Table 5. Here node expansion time is very small, but IDA* tends to make too many expansions, and MREC (M = ∞) appears to be the method of choice.

e) **Travelling Salesman Problem (TSP_G)** [Pearl, 1984, p.90] : The method of solution of the travelling salesman problem described in [Pearl, 1984, p.90] generates a

TABLE 5 : General d_ary Tree

Branching Factor = 9 **Goal Node Depth = 6** **No. of Runs = 100**

algorithm	time in secs		node gen		node exp	
	mean	std	mean	std	mean	std
A*	0.16	0.04	4877.65	1107.50	541.85	123.06
IDA*	3.68	1.99	613901.31	331065.63	68213.94	36785.01
MREC (M=∞)	0.09	0.02	4877.65	1107.50	541.85	123.06

TABLE 6 : TSP_G

Cities = 10 **Mean Cost = 2865.47** **No. of Runs = 100**

algorithm	no. of nodes	time in secs		node gen		node exp	
		mean	std	mean	std	mean	std
A*		0.53	0.38	877.75	335.39	328.33	173.71
MREC (M=∞)		0.52	0.36	873.95	335.96	326.79	173.82
MREC	75	3.80	10.30	10925.02	32889.17	2795.26	8769.20
	150	1.03	1.54	2656.42	4520.58	753.78	1308.01
	300	0.66	0.54	1099.85	748.86	373.93	261.32
	500	0.62	0.50	886.87	360.12	329.06	178.55
	1000	0.62	0.50	873.95	335.96	326.79	173.82

search graph that is not a tree. The method is not efficient, and large instances of the travelling salesman problem cannot be solved, but it was nevertheless selected for study since very few search problems generate search graphs that are not trees. In our implementation, we randomly generated the x and y coordinates of each city, and computed and stored the distance matrix for use by all the algorithms. For the problem illustrated in Table 6, IDA* is not shown because it took a very long time to execute. The heuristic estimate function being consistent, no node is expanded more than once by A*; moreover, node expansion time is appreciable. The running times of A* and MREC (M = ∞) are almost the same, and the running time of MREC increases sharply as M decreases. This is particularly noticeable here because the search graph is not a tree. Here, M can be viewed as the number of nontip nodes (along with their successor lists) that can be stored in the explicit graph. It is easy to express the total number of nodes in the explicit graph as a function of the number of nontip nodes and the branching factor [Sen, 1988].

5 Concluding Remarks

Our findings from the experimental study are summarized below:

1. When M is very small, MREC runs like IDA*, and is suitable for problems such as the 15-puzzle. When M is large, MREC runs as fast as A* on problems for which node expansion time is high. It is thus capable of serving the functions of both A* and IDA*.

2. MREC can take advantage of additional memory and improve its performance by reducing the total number of node expansions when more memory is available.

In ending, we describe another memory constraint algorithm LIMMARK [Sen, 1988]. This non-recursive algorithm is derived from the marking algorithm MarkA [Bagchi and Mahanti, 1985], and uses the idea of *arc marking*. Marking algorithms have the general advantage that they do not need to maintain large lists like OPEN. But a disadvantage of MarkA is that it always requires the entire explicit graph to be stored; moreover, it fails to work if the network has loops. LIMMARK was designed to get around these limitations. In practice, LIMMARK was found to be much slower than IDA* on the 8-puzzle and 15-puzzle problems although it made far fewer node expansions. But the main idea on which it is based appears to be worth consideration : deleting some (nonpromising) nodes and arcs from the explicit graph in order to accommodate newly generated nodes and arcs when memory is in short supply. This idea can

be tried out in certain problem domains. Extension of the same idea to recursive implementations also appears to hold some promise. MREC, as it stands now, makes a virtual expansion of a tip node when memory is unavailable; it is unable to store the graph below the tip node in such a case. Next time when it explores the same node, it is forced to reexpand the node. What could have been done instead is to remove some (nonpromising) nodes and successor arcs from the explicit graph and add the successor list of the newly expanded node to the graph. For example, we can remove nodes and arcs from the least recently explored path. This could, in principle, reduce the number of reexpansions of a node. The strategy will succeed provided we are able to find out nonpromising nodes and arcs easily; an improper choice can lead to oscillations, the algorithm throwing a node out of the explicit graph but entering it again soon after. For the 8-puzzle and 15-puzzle problems, the Manhattan distance function is the commonest heuristic. When such a weak heuristic function is used, all paths in the explicit graph tend to appear equally promising and the method cannot be suitably applied. The idea is nevertheless interesting and deserves further study. A modified version of procedure EXPLORE incorporating the above idea is given below. There is no need here of any virtual expansion of a tip node in UPDATE.

```
procedure EXPLORE(n:node;var bnode:integer);
begin
  if n is a goal node then
    begin terminate := true; return; end;
  if n is a tip node then
    begin
      if sufficient memory not available for
        storing successors of n
      then remove nonpromising nodes and arcs
        from the explicit graph;
      EXPAND(n);
    end;
  UPDATE(n,bnode);
end;
```

A memory constraint algorithm of a somewhat different kind, called MA*, has recently been proposed by Chakrabarti *et al.* [1989]. MA* is based on A*, and uses an OPEN list. Expansions are so controlled that successors get generated and added to OPEN one at a time. Nonpromising nodes get thrown out of OPEN when the memory bound is exceeded. Data on running time of MA* is unavailable, but as in the case of other pruning algorithms like LIMMARK, the overheads are likely to be high. A comparative assessment of MREC and MA* has not been made yet.

The algorithm MREC ($M = \infty$) can be readily generalized to heuristic search in AND/OR graphs with loops (see the recursive algorithm REC_A in [Sen, 1988]). No existing marking algorithms for heuristic search in AND/OR graphs is able to take care of loops. Moreover, all established algorithms for AND/OR graphs are non-recursive in nature. We expect algorithm REC_A to run quite fast; its average performance is likely to be as good as, if not better than, that of existing marking algorithms for AND/OR graphs. It would be easy to reformulate REC_A to run under memory constraints. A detailed theoretical and empirical study of algorithm REC_A is yet to be undertaken. An alternative memory constraint algorithm for AND/OR graphs can be found in [Chakrabarti *et al.*, 1989].

References

[Bagchi and Mahanti, 1985] A. Bagchi and A. Mahanti, Three approaches to heuristic search in networks, *JACM*, vol.32, no.1,January 1985, pp.1-27.

[Chakrabarti *et al.*, 1989], P. P. Chakrabarti, S. Ghose, Arup Acharya and S. C. De Sarkar, Heuristic search in restricted memory, *Artificial Intelligence*, forthcoming.

[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 travelling salesman problem, *Operations Research*, vol.11, 1963, pp.972-989.

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

[Sen, 1988] Anup K. Sen, *Heuristic Search Algorithms : A Theoretical-Cum-Empirical Evaluation*, Ph. D. Thesis, Department of Computer Science, University of Calcutta, November 1988.

[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*, St. Paul, U.S.A., August 1988, pp. 145-149.