Memory-Bounded Bidirectional Search

Hermann Kaindl
Siemens AG Österreich
Geusaugasse 17
A–1030 Wien, Austria — Europe
e-mail: kaih@siemens.co.at

Aliasghar Khorsand
Huglgasse 13-15/6
A–1150 Wien
Austria — Europe

Abstract

Previous approaches to bidirectional search require exponential space, and they are either less efficient than unidirectional search for finding *optimal* solutions, or they cannot even find such solutions for difficult problems. Based on a memory-bounded unidirectional algorithm for trees (SMA^*), we developed a graph search extension, and we used it to construct a very efficient memory-bounded bidirectional algorithm. This bidirectional algorithm can be run for difficult problems with bounded memory. In addition, it is much more efficient than the corresponding unidirectional search algorithm also for finding optimal solutions to difficult problems. In summary, bidirectional search appears to be the best approach to solving difficult problems, and this indicates the extreme usefulness of a paradigm that was neglected for long.

Notation

s, t	Start node and goal node, respectively.
$\Gamma_1(n)$	Successors of node n in the problem graph.
$\Gamma_2(n)$	Parents of node n in the problem graph.
d	Current search direction index; when search is in the forward direction $d = 1$, and when in the backward direction $d = 2$.
d'	$3 - d$; it is the index of the direction opposite to the current search direction.
$g_i^*(n)$	Cost of an optimal path from s to n if $i = 1$, or from t to n if $i = 2$.
$h_i^*(n)$	Cost of an optimal path from n to t if $i = 1$, or from n to s if $i = 2$.
$g_i(n), h_i(n)$	Estimates of $g_i^*(n)$ and $h_i^*(n)$, respectively.
$f_i(n)$	Static evaluation function.
$F_i(n)$	Revised evaluation after pathmax or backup.
C^*	Cost of an optimal path from s to t.
C	Cost of a solution path from s to t found.
$L_{\min}$	Cost of the best (least costly) complete path found so far from s to t.
TREE_1	The forward search tree.
TREE_2	The backward search tree.
OPEN_i	The set of open nodes in TREE_i.
CLOSED_i	The set of closed nodes in TREE_i.
$p_i(n)$	Parent of node n in TREE_i.

Introduction

Originally, bidirectional heuristic search did not work as expected. Although the bidirectional approach was shown to be more efficient than its unidirectional counterpart when heuristic knowledge is unavailable, the inverse result was found in experiments with *BHPA* using a heuristic evaluation function (Pohl 1971). *BS** (Kwa 1989) improved *BHPA* technically, but its performance was only nearly as good as the unidirectional A^* (Hart, Nilsson, & Raphael 1968). There was consensus that bidirectional heuristic search is afflicted with the problem of search fronts missing each other. Consequently, *wave-shaping* techniques were investigated (de Champeaux & Sint 1977; de Champeaux 1983; Davis, Pollack, & Sudkamp 1984; Politowski & Pohl 1984). This work showed that bidirectional heuristic search can be rather efficient in terms of the number of expanded nodes. However, these algorithms are either excessively computationally demanding, or they have no restriction on the solution quality.

Recent results show that the missing of the search fronts is not the central problem (Köll & Kaindl 1993). In fact, the fronts typically meet rather early. However, when aiming for *optimal* solutions, much effort has to be spent for subsequently improving the solution quality, and finally for proving that there is indeed no better solution possible. Therefore, only slightly relaxing the requirements on the solution quality yields strong improvements in efficiency.

In fact, the algorithms presented in (Köll & Kaindl 1993) proved that the bidirectional paradigm is efficient in terms of node expansions when searching for ε-admissible solutions, without using computationally very demanding wave-shaping techniques.[1]

However, while efficient *linear-space* algorithms like *IDA** (Korf 1985) and *RBFS* (Korf 1993) have been developed for the unidirectional case, classical bidirectional search typically requires exponential space. (Usually, two classical *best-first* searches like A^* are used for the opposing search fronts.) In fact, is seems impossible to implement bidirectional search

[1] A search algorithm is called *ε-admissible* if it guarantees that solution costs are bounded by $(1 + \varepsilon)C^*$, i.e., a factor of the cost of an optimal solution (Pearl & Kim 1982; Pearl 1984). This is the same approach as the one of an *ε-approximate* algorithm (Horowitz & Sahni 1978) which shall find approximate solutions to NP-hard problems.

with linear-space requirement, since at least part of one of the search fronts must be in memory in order to recognize meeting of these fronts. Therefore, it would seem that bidirectional search cannot be used for problems of the same difficulty as solvable by linear-space algorithms due to its apparently inherent memory requirement.[2]

In this paper, we show how bidirectional search can be performed very efficiently using *bounded* memory. The key idea is to use a unidirectional algorithm that is memory-bounded by it own — instead of best-first algorithms with exponential memory requirements like A^*. From several such approaches existing today (MA^* (Chakrabarti *et al.* 1989), $MREC$ (Sen & Bagchi 1989), the approach of using certain tables for IDA^* (Reinefeld & Marsland 1991), and ITS (Mahanti *et al.* 1992)), we selected the first one, since it grows the search tree dynamically. Unfortunately, there are several technical problems with the algorithm MA^*. Therefore, we used the improved SMA^* by Russell (Russell 1992) as a basis for our bidirectional algorithms. Actually, we extended it first for directed acyclic graphs (which is not trivial due to its strategy of deleting and re-generating nodes).

While our bidirectional algorithm does not contain additional new techniques, it is a rather complicated integration of several approaches. This integration is necessary for showing an important result: bidirectional heuristic search can be performed efficiently with bounded memory, and it can be more efficient than corresponding unidirectional search also for finding optimal solutions.

First, we describe the development of an ε-admissible graph search extension of SMA^* — we named it $WSMAG^*$ (weighted simplified memory-bounded A^* for graphs). Based on it, we show our construction of bidirectional search with bounded memory — the algorithm we present here is called $MBBS$ (memory-bounded bidirectional search). Then we illustrate the key results and compare them to related work. Finally, we discuss *why* our approach works well.

An ε-Admissible Graph-Search Extension of SMA^*

A short review of SMA^*

SMA^* is a direct successor of MA^*. A key idea of these algorithms is to tradeoff the number of nodes generated and the number of nodes saved. They retain as many nodes as possible, and prefer to retain the most promising ones. As soon as the given memory limit is reached, MA^* prunes all but the ones with the best evaluation (f-cost). In contrast to A^*, these algorithms only partially-expand nodes, generating the successors one at a time.

[2] A bidirectional algorithm sketched in (Korf 1985) utilizes *depth-first iterative-deepening* (without using heuristic knowledge). Still, its space requirement is $O(b^{d/2})$.

Unfortunately, MA^* is quite complicated and very difficult to implement efficiently (Korf 1993). Even worse, it is not correct in the sense that it can return suboptimal solutions (Mahanti *et al.* 1992), although Chakrabarti *et al.* claimed its *admissibility*.

SMA^* improves MA^*, since it preserves information using "pathmax" with the backed-up f-costs (see line 9 in the pseudocode of $WSMAG^*$ in Appendix A), while MA^* loses this information. Moreover, SMA^* maintains fewer f-cost quantities (making it simpler), and it backs up values once per fully-expanded node, rather than once per node generated (reducing the overhead). Finally, SMA^* adds and prunes only one node at a time (saving re-generations of nodes).

Unfortunately, the pseudocode in (Russell 1992) contains some few bugs.[3] An improved (and more formal) pseudocode of SMA^* can be found in Appendix A, when omitting lines 10 and 11, and using $\varepsilon = 0$ in line 8. Moreover, SMA^* as presented in (Russell 1992) only deals with the case of finding *optimal* solutions and *tree* search.

ε-admissibility

The extension of SMA^* to an ε-admissible version is analogous to the corresponding extensions of IDA^* to $WIDA^*$ or of A^* to WA^* (which we call HPA^* for historic reasons) (Korf 1993). It is just necessary to multiply the heuristic component h of the usual A^*-type evaluation by a weight. According to the notation in (Pearl & Kim 1982), we use the weight $(1 + \varepsilon)$ (see line 8 of $WSMAG^*$ in Appendix A). We named this ε-admissible algorithm $WSMA^*$.

Graph search

The extension of SMA^* from searching trees to dealing with directed acyclic graphs is less trivial. In A^*, this issue is usually dealt with by simply moving a repeatedly found node from CLOSED back to OPEN. Such simple attempts for a straight-forward solution can lead to infinite loops due to SMA^*'s strategy of deleting and re-generating nodes (Khorsand 1994). The best way to solve this issue that we found uses a technique we call *blocking* (see the pseudocode of GRAPH-CONTROL in Appendix A). Whenever a new path to a node already stored is found, a distinction is made whether the new path is better than the old one. If this is not the case, the arc from the parent on the new path

[3] *best* ← deepest least-f-cost leaf in OPEN;
should be ... node ... instead of ... leaf ..., since each partially expanded node should be selectable for further expansion;
delete shallowest, highest-f-cost node in OPEN;
should be ... leaf ... instead of ... node ..., since only nodes without any generated successor can be deleted safely;
the procedure BACKUP should have the structure given in Appendix A; otherwise no backup of the value could occur to the root.

is permanently deleted, otherwise the one from the parent on the old path. Moreover, when the new path is better, the subtree previously grown from this node is pruned. In both cases it must be checked whether the other successors of the respective parent of the common node are also blocked. If this is the case and this parent node is already completed, then also this node is deleted, and the blocking mechanism is performed recursively (see SUCC-CHECK in Appendix A).

We call the resulting (ε-admissible) algorithm for directed acyclic graphs $WSMAG^*$ (for details see (Khorsand 1994)). The complete pseudocode is given in Appendix A.

Bidirectional Heuristic Search with Bounded Memory

The classical approaches to implementing bidirectional search typically use A^*- or HPA^*-type search from both sides concurrently. On a single processor, of course, only one direction can be followed at a time. For the selection of search direction, (Pohl 1971) proposed the so-called cardinality criterion (see line 4 of $MBBS$ in Appendix B).

Since this approach runs into trouble when the memory is limited, we use instead the memory-bounded $WSMAG^*$ algorithm from both sides. In the following, we sketch the development of an efficient memory-bounded bidirectional algorithm based on this idea.[4]

First, the two search fronts using $WSMAG^*$ must meet each other. Since both these dynamically grown trees are stored, this meeting can be efficiently checked using hashing. However, meeting just means that any solution was found. Therefore, it may be necessary to continue the search until a solution of required quality is detected. Even if that is the case, the search may not yet know it.

The *termination condition* of a heuristic bidirectional search is usually based on the best heuristic estimates in both search fronts. When dealing with ε-admissible search, an improved termination condition as introduced in IBS^*_ε (Köll & Kaindl 1993) can be used (see line 3 of $MBBS$ in Appendix B). Since for checking this termination condition the $g+h$-values are needed (in addition to the F-costs used by SMA^*), the backup procedure was extended accordingly.

Moreover, it is useful to consider whether and how the technical improvements of BS^* (Kwa 1989) to $BHPA$ are also applicable in our approach.[5] Because of the partial-node-expansion strategy of SMA^* as well as its method of deleting and re-generating nodes, there are significant differences to BS^* that do not allow the use of its *nipping* and *pruning* technique in our approach. However, *trimming* and *screening* are very useful especially for the task of finding optimal solutions, and these techniques fit well into the paradigm of deleting nodes by SMA^* (see lines 14–16 and 31–33 of $MBBS$ in Appendix B). However, similar to the extension of dealing with graphs, it is necessary here to use the blocking technique described above. The parents of the currently best node (in the case of screening) or of all the trimmed nodes must be checked whether their other successors are also blocked (see lines 16 and 33).

The pseudocode of $MBBS$ is given in Appendix B. Its subroutines are analogous to those of $WSMAG^*$.

Results

Given an admissible heuristic h, both $WSMAG^*$ and $MBBS$ are ε-admissible, i.e., if a path exists from s to t, they terminate with a solution whose cost does not exceed the optimal cost by more than a factor $(1+\varepsilon)$. Due to lack of space we cannot include here the formal proofs (see (Khorsand 1994)).

Below we summarize the empirical results of extensive experiments on an NP-complete problem (finding optimal solutions to the 15-Puzzle) and on the relaxed problem of finding near-optimal solutions (with a guaranteed worst-case bound). Fig. 1 compares $WSMAG^*$ and $MBBS$ (storing 256k nodes) with several algorithms.[6] The x-axis represents various values of ε, the given worst-case bound as guaranteed by these algorithms. The comparison is in terms of generated nodes as, e.g., in (Köll & Kaindl 1993; Korf 1993), and the y-axis shows the average number of node generations on the 15-Puzzle on a logarithmic scale.

This figure and statistical tests (for details see (Khorsand 1994)) demonstrate the following key results:

- *Significant superiority of $MBBS$ to (unidirectional) linear-space search in terms of generated nodes*
  On the whole range, $MBBS$ generates much fewer nodes than $WIDA^*$ and $RBFS$. Therefore, $MBBS$ makes excellent use of the modest amount of memory given — 256k nodes: 2 orders of magnitude more

[4] Actually, we developed several bidirectional algorithms along the lines of (Köll & Kaindl 1993), but we focus on the most efficient one in this paper. The other algorithms and details are described in (Khorsand 1994).

[5] These improvements are the following:
(i) *nipping*: When a node is selected for expansion which is already closed in the opposite search tree, it can just be closed *without* expansion.
(ii) *pruning*: In the same situation, descendants of this node in the opposing OPEN list can be removed.
(iii) *trimming*: Open Nodes (in both directions) whose f-values are $\geq L_{min}$ can be removed.
(iv) *screening*: Nodes whose f-values are $\geq L_{min}$ need not to be stored in the OPEN lists.

[6] We have used the set of 100 instances from (Korf 1985). The data in the figure currently lack the results of the 11 most difficult problems, since these experiments are for some of the algorithms still running at the time of this writing. Since HPA^* and IBS^*_ε cannot solve all these instances for all values of ε even when storing millions of nodes, their corresponding lines are not complete in this figure.

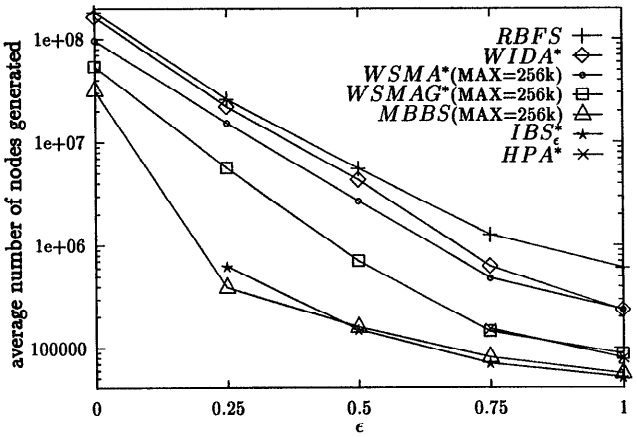

Figure 1: Comparison on the 15-Puzzle.

nodes are generated in the average for finding optimal solutions. However, for the task of finding optimal solutions ($\varepsilon = 0$), *WIDA** is still faster on this puzzle (see below).

- *Significant superiority of the novel bidirectional approach to the corresponding unidirectional one (using the same amount of memory)*
  Also for finding optimal solutions *MBBS* is significantly better than both *WSMA** and *WSMAG**. Generally, the improvement is even slightly stronger comparing the run time, since instead of one priority queue the bidirectional algorithm uses two shorter ones.
- *Significant superiority of the memory-bounded bidirectional search to classical (unidirectional) best-first search (HPA*)*
  The improvement in terms of node generations is statistically significant, although *MBBS* stores much fewer nodes. For $\varepsilon \leq 0.5$, *HPA** cannot even find solutions to all the problems even when storing millions of nodes.
- *Comparable results of the memory-bounded to more classical bidirectional search (IBS^*_ε), that uses much more memory*
  The memory-bounded approach is even significantly better for $\varepsilon = 0.25$ due to its partial node expansions.

Despite Russell's improvements to *MA**, an efficient implementation of these memory-bounded algorithms is non-trivial (see (Khorsand 1994) for the data structures used instead of the binary trees of binary trees suggested by Russell, which are not efficient for the (usual version of) the sliding-tile puzzles with uniform-cost evaluation). Unfortunately, it strongly depends on the domain and also the efficiency of implementation, whether the overhead of maintaining the priority queues is deteriorating the performance or not. In particular, the importance of the overhead depends on the effort for computing heuristic values. Even the machine architecture can influence the relative running time when there are differences in the size of memory used. While node generation and evaluation is very efficient for the sliding-tile puzzles (Korf 1993), we primarily wanted to compare the algorithms on a basis that is more independent of such factors.

Compared to the worst-case bound ε, the *average* solution quality is much better for all the algorithms compared here. For $\varepsilon = 0.25$, e.g., the cost C of a solution found must be ≤ 1.25 times the cost of an optimal solution C^*. In the average, *MBBS* finds solutions of quality $C = 1.04C^*$ here, i.e., they are only 4 percent worse than optimal ones, which require about eighty times more node generations to be found.

Related Work

Apart from the genesis of *MBBS* based on the unidirectional *A**, *MA** and *SMA**, as well as the bidirectional *BHPA*, *BS** and IBS^*_ε, there are some relations to other *unidirectional* search algorithms with reduced space requirements. We focus here on their results on the sliding-tile puzzles:

- *MREC* (Sen & Bagchi 1989) did not achieve a real improvement over *IDA**, partly because they used a *tree* version; (Korf 1993) reports that the number of node generations of a *graph* version storing 100k nodes reduced the number of node generations by 41 percent compared to *IDA** on the 15-Puzzle;
- *ITS* (Mahanti *et al.* 1992) generated the same number of nodes as *IDA** on the 15-Puzzle;
- the best version of the approach of using certain tables for *IDA** (Reinefeld & Marsland 1991) examined 45.82 percent of the nodes generated by pure *IDA**, storing 256k nodes.

In summary, all these algorithms are less efficient than our bidirectional search algorithm *MBBS* even for finding *optimal* solutions (at least on the difficult 15-Puzzle and in terms of node generations). Moreover, *MBBS* can find near-optimal solutions with a guaranteed worst-case bound, a task on which it is even much more efficient.

Discussion

An interesting question is how this bidirectional approach can be better than its unidirectional counterpart. In an exponential search space, bidirectional search has the potential to divide the exponent by 2. However, the first results with bidirectional *heuristic* search were bad (Pohl 1971), and the explanation given there in terms of the *missile metaphor* was misleading. The primary issue appears to be that after the first meeting especially a bidirectional algorithm aiming for *optimal* solutions has to spend much effort for subsequently improving the solution quality, and finally for proving that no other open node can give a better solution (Köll & Kaindl 1993). Therefore, only a slight relaxation of solution quality already leads to strong improvements in efficiency, and in particular to even

more than in the unidirectional case. Actually, the results of $MBBS$ for $\varepsilon = 0.25$ are relatively much better than those for $\varepsilon = 0$.

However, Kwa (Kwa 1989) already noted the relative improvement of BS^* versus A^* with increasing problem difficulty (in the context of finding optimal solutions). This tendency was also observed in the case of finding near-optimal solutions (Köll & Kaindl 1993). Consequently, the dynamic memory utilization of $MBBS$ (based on SMA^*) allows it to perform the bidirectional search efficiently for problems that are difficult enough to make it better than its unidirectional counterpart.

Conclusion

In summary, we developed a graph search extension of SMA^*, and we used this unidirectional algorithm to construct a very efficient memory-bounded bidirectional algorithm. This algorithm can use as much memory as there is available (a constant amount).

The construction of this algorithm uses primarily known techniques (apart from the graph search extension with its novel blocking mechanism). However, this combination is necessary for showing an important result: bidirectional heuristic search can be performed efficiently with bounded memory, and it can be more efficient than its unidirectional counterpart also for finding *optimal* solutions.

If problems are too difficult for finding optimal solutions within reasonable time, finding near-optimal (ε-admissible with a small ε) solutions may help. For this task the bidirectional approach is even better.

Acknowledgments

We would like to thank Dennis de Champeaux, Gerhard Kainz, and Ira Pohl for comments on earlier versions of this paper. Our implementations are based on the code of WA^* provided by Richard Korf.

APPENDIX A: Pseudocode of WSMAG*

```
procedure WSMAG*(s, t);
 1. OPEN ← {s}; CLOSED ← ∅; USED ← 1;
 2. while ( OPEN ≠ ∅ ) and (no solution found) do
        /* select deepest node with lowest F-value : */
 3.     B ← {x | (x ∈ OPEN) ∧ (∀y ∈ OPEN: F(x) ≤ F(y))};
                        /* set of nodes with lowest F-value */
 4.     select best ∈ {x | x ∈ B ∧ (∀y ∈ B: g(x) ≥ g(y))};
                                    /* deepest node from B */
 5.     if best ≠ t then   /* best is not goal */
 6.         succ ← next-successor ∈ Γ(best);
 7.         g(succ) ← g(best) + c(best, succ);
 8.         f(succ) ← g(succ) + (1+ε)·h(succ);
 9.         F (succ)← max(F(best), f(succ));        /* pathmax */
10.         if succ ∈ TREE then          /* succ already stored */
11.             GRAPH-CONTROL;
            else
12.             MEM-CONTROL;
13.             OPEN ← OPEN ∪ {succ}; USED ← USED + 1;
                                    /* insert succ in OPEN */
14.             S(best) ← S(best) ∪ {succ};
                        /* insert succ in best´s successor list */
            endif
15.         if completed(best) then
16.             BACKUP(best);
            endif
17.         if Γ(best) all in memory then
18.             OPEN ← OPEN \ {best};
19.             CLOSED ← CLOSED ∪ {best};       /* close best */
            endif
        endif
    endwhile
    endprocedure

procedure MEM-CONTROL;
    /* if memory is full, delete a node */
 1. if USED = MAX then
        /* delete shallowest, highest- F-value leaf in OPEN ,
           a leaf node has no successor either in OPEN or CLOSED : */
 2.     W ← {x | (x ∈ OPEN) ∧ (∀y ∈ OPEN: F(x) ≥ F(y))};
 3.     select worst ∈ {x | x ∈ W ∧ (∀y ∈ W: g(x) ≤ g(y)) ∧ (S(x) = ∅)};
 4.     delete worst;
 5.     USED ← USED - 1; /* remove worst from its parent´s succ. list */
 6.     S(p(worst)) ← S(p(worst)) \ {worst};
 7.     if p(worst) ∈ CLOSED then
 8.         CLOSED ← CLOSED \ {p(worst)};
 9.         OPEN ← OPEN ∪ {p(worst)};
        endif
    endif
endprocedure

procedure BACKUP(n)
    /* back up F-values */
 1. if completed(n) then
        /* least F-value of all successors: */
 2.     newF ← F(x) | x ∈ Γ(n) ∧ (∀y ∈ Γ(n): F(x) ≤ F(y));
 3.     if newF > F(n) then
 4.         F(n) ← newF;
 5.         reorder OPEN according to new F-value;
 6.         if n has a parent then
 7.             BACKUP(p(n));
            endif
        endif
    endif
endprocedure

procedure GRAPH-CONTROL;
    /* deal with directed acyclic graphs */
 1. old ← old node which is equal to succ ;
 2. if g(succ) ≥ g(old) then      /* new path is not better */
 3.     block succ in best's successor list;
 4.     SUCC-CHECK(best);
    else       /* a better path to old has been found */
 5.     prune subtree(old);
    /* remove all nodes which are in subtree with root old */
 6.     block old in p(old)'s successor list;
 7.     USED ← USED - |subtree (old)|;
 8.     SUCC-CHECK(p(old));
 9.     OPEN ← OPEN ∪ {succ}; USED ← USED + 1;
                                    /* insert succ in OPEN */
10.     S(best) ← S(best) ∪ {succ}; /* insert succ in best´s succ. list */
    endif
endprocedure

procedure SUCC-CHECK(n)
    /* checks whether the other successors are also "blocked" */
 1. if completed(n) and n has only "blocked" successors then
 2.     delete n; USED ← USED - 1;
 3.     block n in p(n)'s successor list;
```

```
4.      if n has a parent then
5.          SUCC-CHECK(p(n));
        endif
    else
6.      BACKUP(n);
    endif
endprocedure
```

APPENDIX B: Pseudocode of MBBS

```
procedure MBBS(s, t);
1.  OPEN_1 ← {s}; OPEN_2 ← {t}; CLOSED_1 ← CLOSED_2 ← ∅;
2.  USED_1 ← USED_2 ← 1; L_min ← ∞;
3.  while (OPEN_1 ≠ ∅) and (OPEN_2 ≠ ∅)
          and (L_min > (1+ε)·max(ghmin_1, ghmin_2)) do
4.      if |OPEN_1| ≤ |OPEN_2| then        /* cardinality criterion */
5.          d ← 1; else d ← 2;
        endif
6.      d' ← 3 - d;        /* set the opposite search direction */
7.      Trimflag ← false;
        /* select deepest node with lowest F-value : */
8.      B ← {x | (x ∈ OPEN_d) ∧ (∀y ∈ OPEN_d: F_d(x) ≤ F_d(y))};
                                /* set of nodes with lowest F-value */
9.      select best ∈ {x | x ∈ B ∧ (∀y ∈ B: g_d(x) ≥ g_d(y))};
                                /* deepest node from B */
10.     succ ← next-successor ∈ Γ_d(best);
11.     g_d(succ) ← g_d(best) + c(best, succ);
12.     f_d(succ) ← g_d(succ) + (1+ε)·h_d(succ);
13.     F_d(succ) ← max(F_d(best), f_d(succ)); /* pathmax */
14.     if g_d(succ) + h_d(succ) ≥ L_min then      /*screening */
15.         block succ in best's successor list;
16.         SUCC-CHECK(best);
        else
17.         if succ ∈ TREE_d then        /* succ already stored */
18.             GRAPH-CONTROL;
            else
19.             MEM-CONTROL;
20.             OPEN_d ← OPEN_d ∪ {succ}; USED_d ← USED_d + 1;
                                /* insert succ in OPEN */
21.             S_d(best) ← S_d(best) ∪ {succ};
                        /* insert n succ best's successor list */
            endif
22.         if succ ∈ TREE_d' and g_1(succ) + g_2(succ) < L_min then
                                /* better solution found */
23.             L_min ← g_1(succ) + g_2(succ);        /* update L_min */
24.             MeetingNode ← succ;
25.             Trimflag ← true;
            endif
        endif
26.     if completed(best) then
27.         NEW-BACKUP(best);
        endif
28.     if Γ_d(best) all in memory then
29.         OPEN_d ← OPEN_d \ {best};
30.         CLOSED_d ← CLOSED_d ∪ {best};        /close best */
        endif
31.     if Trimflag then        /* trimming */
32.         remove from OPEN_1 and OPEN_2 those nodes n with
            g(n) + h(n) ≥ L_min and which have no successors and
            are not source or meeting nodes; block them in their
            parent's successor list;.
33.         SUCC-CHECK(p(n));    /* parents of all removed nodes */
        endif
    endwhile
endprocedure
```

References

Chakrabarti, P.; Ghose, S.; Acharya, A.; and DeSarkar, S. 1989. Heuristic search in restricted memory. *Artificial Intelligence* 41(2):197–221.

Davis, H.; Pollack, R.; and Sudkamp, T. 1984. Towards a better understanding of bidirectional search. In *Proc. of AAAI-84*, 68–72. Austin, TX: Los Altos, CA.: Kaufmann.

de Champeaux, D., and Sint, L. 1977. An improved bidirectional heuristic search algorithm. *J. ACM* 24:177–191.

de Champeaux, D. 1983. Bidirectional heuristic search again. *J. ACM* 30:22–32.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics (SSC)* SSC-4(2):100–107.

Horowitz, E., and Sahni, S. 1978. *Fundamentals of Computer Algorithms.* New York: Springer-Verlag.

Khorsand, A. 1994. Heuristische Graph-Suche mit begrenzbarem Fehler und Speicherbedarf sowie Einheitskosten. Diplomarbeit, Technische Universität Wien.

Köll, A., and Kaindl, H. 1993. Bidirectional best-first search with bounded error: Summary of results. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, 217–223.

Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Kwa, J. 1989. BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artificial Intelligence* 38(2):95–109.

Mahanti, A.; Nau, D.; Ghosh, S.; and Kanal, L. 1992. An efficient threshold heuristic tree search algorithm. Technical Report UMIACS TR 92-29, CS TR 2853, Computer Science Department, University of Maryland, College Park, Md.

Pearl, J., and Kim, J. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 4(4):392–399.

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

Pohl, I. 1971. Bi-directional search. In *Machine Intelligence 6*, 127–140. Edinburgh: Edinburgh University Press.

Politowski, G., and Pohl, I. 1984. D-node retargeting in bidirectional heuristic search. In *Proc. of AAAI-84*, 274–277. Austin, TX: Los Altos, CA.: Kaufmann.

Reinefeld, A., and Marsland, T. 1991. Memory functions in iterative-deepening search. Technical Report FBI-HH-M-198/91, Fachbereich Informatik, Universität Hamburg, Hamburg, Germany.

Russell, S. 1992. Efficient memory-bounded search methods. In *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, 1–5. Vienna, Austria: Chichester: Wiley.

Sen, A., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proc. Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, 297–302.