A Case Study of Revisiting Best-First vs. Depth-First Search

Andreas Auer[1] and **Hermann Kaindl**[2]

Abstract. Best-first search usually has exponential space requirements on difficult problems. Depth-first search can solve difficult problems with linear space requirements, but it cannot utilize large additional memory available on today's machines. Therefore, we revisit the issue of when best-first or depth-first search is preferable to use. Through algorithmic improvements, it was possible for the first time to find optimal solutions of certain difficult problems (the complete benchmark set of Fifteen Puzzle problems) using traditional best-first search (with the Manhattan distance heuristic only). Our experimental results show that this search can solve them overall faster than any of the previously published approaches (using this heuristic). Note that this search approach was believed to be incapable of solving randomly generated instances of the Fifteen Puzzle within practical resource limits because of its exponential space requirements. So, our case study suggests that changes in hardware and algorithmic improvements together can revise the previous assessment of best-first search.

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'	$3 - d$; index of the opposite search direction.
$c_i(m, n)$	Cost of the direct arc from m to n if $i = 1$, or from n to m if $i = 2$.
$k_i(m, n)$	Cost of an optimal path from m to n if $i = 1$, or from n to m if $i = 2$.
$g_i^*(n)$	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: $g_i(n) + h_i(n)$.
C^*	Cost of an optimal path from s to t.
$L_{\min}$	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.
$\lvert\text{OPEN}_i\rvert$	Number of nodes in OPEN_i.
CLOSED_i	The set of closed nodes in TREE_i.
$p_i(n)$	Parent of node n in TREE_i.
$p_d^i(n)$	$\underbrace{p_d(p_d \ldots (p_d}_{i-times}(n)) \ldots)$.
Ω_m	The set of nodes in $\text{OPEN}_{d'}$ which are descendants of m in $\text{TREE}_{d'}$.
MeetingNode	Node where TREE_1 met TREE_2 and yielded the best complete path found so far.

[1] Ulmenstr. 5, A-3032 Eichgraben, Austria
[2] Vienna Univ. of Technology, Inst. of Computer Technology, Vienna, Austria

1 INTRODUCTION

Whenever sufficient memory is available, traditional best-first search may be the choice, since it expands the fewest nodes among all admissible algorithms using the same cost function [14]. However, linear-space search algorithms such as DFBB (depth-first branch-and-bound) [9] or IDA* (iterative-deepening-A*) [5] can solve much more difficult problems because they face no real space limitations. But they have large search overheads if there are many distinct cost values, and if the problem graph is not a tree. Since machines with larger and larger memories are becoming available, best-first search can be applicable for many problems in practice (e.g., route planning). Can it solve problems now for which it was believed to be incapable of previously?

In this paper, we investigate this question in a case study using the domain of Fifteen Puzzle problems, which involve 15 sliding tiles in a frame of 4×4 positions. The state space contains $16!/2 \approx 10^{13}$ puzzle configurations. More precisely, we used the standard benchmark of 100 problem instances generated randomly and given in [5]. All the compared algorithms had no domain-specific knowledge about the puzzle available other than the Manhattan distance heuristic. With this heuristic, IDA* was able to solve all 100 problem instances in the sense of finding optimal solutions already in the early eighties. In contrast, A* with 30,000 nodes of storage was not able to solve a single one as reported in [5].

Having a machine with 2 GBytes main storage available meanwhile, we replicated this experiment by giving A* 43 million nodes. Now it can solve 78 problem instances, but most of these are relatively easy, while it still cannot solve the more difficult ones. Instead of trying to get more and more main storage so that A* may be able to solve them all, we found it more interesting and challenging to achieve the result in question through search improvements.

With much improved heuristic functions, much more efficient searches result [1, 6] and even solving Twenty-Four Puzzle problems has become feasible [6, 7]. This, however, signifies improvements of the *knowledge* made available to the search, which is an orthogonal approach. It is well known from the early days of knowledge-based systems that improvements of the knowledge provided by humans can improve performance tremendously. We were not particularly interested in these puzzles per se or in designing better heuristic knowledge for them. It took a few decades until excellent (and still admissible) heuristics were devised for them. In practice it will not always be possible to develop such good heuristics before solutions are required.

Another possibility for improvements is to let the search itself dynamically improve heuristic values according to [3]. Using this approach, a given and less perfect static heuristic can be improved by the machine itself. The current paper contributes by showing how this approach can be utilized well in traditional best-first search.

As indicated above, however, A* still does not perform well even with much larger available storage, and it is in a certain sense optimal over its unidirectional competitors (see [2]). So, we decided to study *bidirectional* heuristic search in this regard. A typical representative of traditional bidirectional heuristic search with "front-to-end" evaluations is BS* [8]. We found that it can solve 95 problem instances on the 2 GByte machine. Those instances that it still cannot solve are much more difficult than the ones it can.

A related algorithm based on dynamic improvements of the heuristic named Max-Switch-A* [4] was reported to solve 79 of the given puzzle instances with 256 MBytes of storage. On the machine with 2 GBytes, we found that it can solve 99, but it still cannot solve the single most difficult problem in the given set.

Is there a better utilization of this approach in traditional best-first search? Unfortunately, due to the dynamic nature of this improved heuristic, its application in such a bidirectional algorithm poses difficult technical problems. Finally, however, we found a simple and elegant way of using this heuristic in an algorithmic improvement of BS*. This combination is able to solve the complete benchmark set on the given machine with 2 GBytes. According to our best knowledge, it can solve it overall faster than any of the previously published approaches using the Manhattan distance heuristic.

The remainder of this paper is organized as follows. In order to make it self-contained, we review some background material on the bidirectional heuristic search algorithm BS* and on the dynamically improved heuristic. Then we elaborate on the problems involved in their combination and some concrete approaches. Finally, we demonstrate the efficiency of the best combination we found in comparison with the best competitors through presenting experimental results.

2 BACKGROUND

First, let us be precise on what constitutes a bidirectional search. When there is one goal node t explicitly given, such a search proceeds both in the forward direction from the start node s to t and in the backward direction from t to s [11]. Bidirectional search is possible if for a given node n the set of parent nodes p_i can be determined for which there exist operators that lead from p_i to n. Searching backwards means generating parent nodes successively from the goal node t. In other words, backward search implements *reasoning* about the operators in the backward direction.

Bidirectional search also works correctly in cases where the costs of inverse arcs between any two nodes are different: the backward search implements reasoning in the backward direction but takes account of the cost of going in the forward direction. More formally, $k_1(m,n) = k_2(n,m)$ is the cost of an optimal path from m to n. So, $k_2(m,n)$ is the cost of an optimal path from n to m, and is used for notational convenience only. All the bidirectional search algorithms dealt with in this paper work correctly under these conditions and do *not* require that the operators are reversible or that the cost of a path is the same in either direction.

In this paper we focus on the kind of traditional bidirectional search with "front-to-end" evaluations: the heuristic evaluation functions $f_d(n)$ estimate the cost of an optimal path to the appropriate endpoint (i.e., $f_1(n)$ uses t as the target for the forward search, and $f_2(n)$ uses s as the target for the backward search). We can view such a search essentially as two A*-type searches in opposite directions, but it has a more subtle termination condition for guaranteeing optimal solutions. These searches in opposite directions are performed quasi-simultaneously, i.e., on a sequential machine one node is expanded after another, but the search direction is changed at least from time to time. The decision for searching in the forward or backward direction is usually made anew for each node expansion, most often according to the *cardinality criterion* [11]: search in the direction with fewer open nodes.

```
   procedure GenericBS*(s, t)
1.    g_1(s) ← g_2(t) ← 0; f_1(s) ← f_2(t) ← h_1(s); L_min ← ∞;
2.    OPEN_1 ← {s}; OPEN_2 ← {t};
3.    CLOSED_1 ← CLOSED_2 ← Ø;
4.    until OPEN_1 = Ø or OPEN_2 = Ø do
5.       <Determine the search direction and set d>;
6.       d' ← 3 − d; /* set the opposite search direction */
7.       <Select m ∈ OPEN_d>;
8.       OPEN_d ← OPEN_d \ {m};
9.       CLOSED_d ← CLOSED_d ∪ {m};
10.      if m ∈ CLOSED_d'
            then /* nip m in TREE_d and prune TREE_d' */
11.            Ω_m ← {n | n ∈ Γ_d'(m) ∧ p_d'(n) = m};
12.            OPEN_d' ← OPEN_d' \ Ω_m;
            else
13.            EXPAND(m);
         endif
      enduntil
14.   if L_min = ∞
         then no path exists
         else
15.         the solution path with cost L_min is
            (s, ..., p_1^2(MeetingNode), p_1(MeetingNode),
            MeetingNode, p_2(MeetingNode), ..., t).
      endif
   endprocedure.
```

Figure 1. A generic BS* algorithm.

The typical representatives of traditional bidirectional heuristic search with "front-to-end" evaluations are the two algorithms BHPA [11] and BS* [8]. In order to be more precise, we developed a pseudocode formulation of a generic BS* algorithm shown in Fig. 1. It is generic in containing a few statements and conditions with possible variations, which are indicated by numbers in bold face. In order to make the pseudocode more readable, its procedure for node expansion is given separately in Fig. 2.

One statement with variations is number 5 in procedure *GenericBS** (see Fig. 1). The original BS* algorithm [8] uses the above-mentioned cardinality criterion here, more precisely:

if $|\text{OPEN}_1| \leq |\text{OPEN}_2|$ **then** $d \leftarrow 1$ **else** $d \leftarrow 2$.

Another generic statement is number 7 in procedure *GenericBS**. Much as usual in traditional best-first search, BS* selects one of the nodes with lowest f_d-value from OPEN_d.

While BHPA explores part of the search space twice, BS* can avoid this due to the following improvements over BHPA:

- *nipping*:
  When a node is selected for expansion which is already closed in

```
    procedure EXPAND(m)
1.      TrimFlag ← false;
2.      foreach n ∈ Γ_d(m) do
3.          g ← g_d(m) + c_d(m, n);
4.          if not <screening condition>
                then /* process node n */
5.              f ← g + h_d(n);
6.              if n ∉ TREE_d /* n is a new node */
                    then
7.                  g_d(n) ← g; f_d(n) ← f; p_d(n) ← m;
8.                  OPEN_d ← OPEN_d ∪ {n};
9.              elsif g < g_d(n) /* better path to n found */
                    then
10.                 g_d(n) ← g; f_d(n) ← f; p_d(n) ← m;
11.                 if n ∈ CLOSED_d
                        then /* reopen n */
12.                     CLOSED_d ← CLOSED_d \ {n};
13.                     OPEN_d ← OPEN_d ∪ {n};
                    endif
                endif
14.             if n ∈ TREE_d' and g_1(n) + g_2(n) < L_min
                    then /* update L_min */
15.                 L_min ← g_1(n) + g_2(n);
16.                 MeetingNode ← n;
17.                 TrimFlag ← true;
                endif
            endif
        endforeach
18.     if TrimFlag
            then /* trim the open lists */
19.         Remove from OPEN_1 and OPEN_2 those nodes n
            satisfying the <trimming condition> and which
            are not source nodes (for OPEN_1 the source node
            is s; for OPEN_2 it is t)
        endif
    endprocedure.
```

Figure 2. The procedure for node expansion.

the opposite search tree, BS* closes it *without* expansion.

- *pruning*:
  In the same situation, BS* removes descendants of this node in the opposite OPEN_{3-d} list.

In addition, BS* uses the formula

$$f_d(n) = g_d(n) + h_d(n) \geq L_{min} \tag{1}$$

for removing nodes from the OPEN_d lists:

- *trimming*:
  When a new solution with reduced cost L_{min} is found, BS* removes all those open nodes (in both directions) whose f_d-values satisfy (1). This is the <trimming condition> of statement number 19 in procedure *EXPAND* (see Fig. 2).
- *screening*:
  If the f_d-value of a newly generated node satisfies (1), BS* does not place it in the OPEN_d list at all. In fact, formula (1) is also the <screening condition> of statement number 4 in procedure *EXPAND*.

BS* terminates when OPEN_1 or OPEN_2 is empty. BS* is *admissible* if h_d is *consistent*.[3] We assume the availability of a consistent static heuristic evaluation function h_d in both directions.

In this paper, we also make use of the *Max* heuristic [3]. For a given node A to be expanded in the forward direction $d = 1$, it is known based on the search already undertaken to A, how much the actual path is more costly than its heuristic estimate:

$$\mathit{Diff}_2(A) = g_1(A) - h_2(A) \tag{2}$$

Moreover, $fmin_2$ is the minimal f_2-value of all open nodes in the opposite search frontier. As proven and illustrated in [3], adding these two components does not overestimate an optimal path through A. Since such a dynamic estimate may also be smaller than the static heuristic, the maximum of the two is taken:

$$F_1(A) = \max(f_1(A), fmin_2 + \mathit{Diff}_2(A)) \tag{3}$$

The resulting function F_1 is admissible but not consistent.

3 HOW TO USE A DYNAMICALLY IMPROVED HEURISTIC IN A BS*-LIKE ALGORITHM

The first and immediately obvious use of F_d — the general version of F_1 as given in (3) — is instead of f_d for sorting the nodes in the OPEN_d lists for expansion. However, there are a few technical problems involved with incorporating the *Max* method into BS*, two of which we name here. First, the search fronts meet later, since the use of F_d has the side effect of delayed expansion of those nodes with a high chance of meeting the other search frontier. This is also a reason why algorithms with the *Max* heuristic do not *dominate* their counterparts without. A bigger problem is that BS* requires a consistent heuristic because of nipping and pruning. Even worse, the usual trick to make an admissible heuristic consistent (using the maximum of the heuristic values on a path) cannot be applied, since $fmin_d$ may increase dynamically *during* the search.

Still, we have been able to devise several admissible algorithms based on BS* that make use of the *Max* heuristic. We sketch here two selected algorithms:

- Max-BS*
  The first algorithm is a simple modification of BS*. In fact, only statement number 7 in procedure *GenericBS** has to be instantiated differently to the original BS* algorithm. As indicated above, the modified algorithm uses F_d instead of f_d for sorting the nodes in the OPEN_d list for expansion. But it does so only in *one* direction, i.e., d is fixed. This allows nipping and pruning from one side, still avoiding exploration of the search space twice.
- BiMax-BS$^*_F$
  This algorithm utilizes the *Max* heuristic in *both* search directions concurrently, i.e., really bidirectionally. However, it does it in a completely different way than Max-BS*. In particular, BiMax-BS$^*_F$ does *not* use F_d for node selection, but f_d like the original BS* algorithm. In contrast, BiMax-BS$^*_F$ uses this dynamically improved heuristic for trimming and screening. More precisely, it uses the following formula as the <trimming condition> of statement number 19 and as the <screening condition> of statement number 4 in procedure *EXPAND*, and for either value of d:

$$F_d(n) = \max(f_d(n), fmin_{d'} + \mathit{Diff}_{d'}(n)) \geq L_{min} \tag{4}$$

[3] h_d is said to be consistent if $h_d(m) \leq h_d(n) + k_d(m, n)$ for all nodes m and n. If for any goal node t $h_d(t) = 0$, this implies that h_d is admissible, i.e., the heuristic function never overestimates the minimal cost.

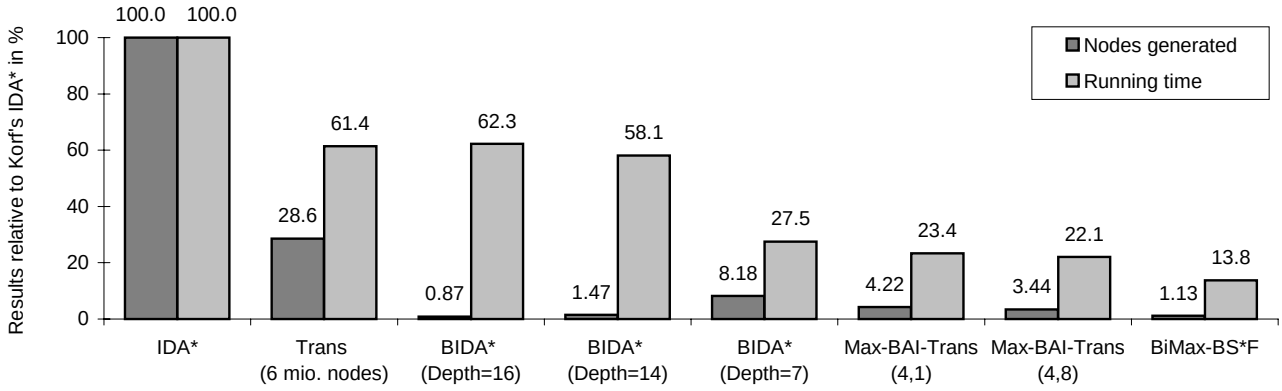

Figure 3. Comparison on the complete benchmark set of Fifteen Puzzle problem instances.

This formula essentially combines formula (1) with the generalized version of (3). Its application improves both the number of nodes expanded and the memory utilization of BS* through discarding many more nodes even before they are stored (screening), due to the dynamically improved heuristic of the *Max* method. In effect, it prunes many nodes which otherwise would have to be expanded. Since BiMax-BS$^*_F$ uses the consistent heuristic f_d for sorting the nodes in both OPEN$_d$ lists (instead of the admissible but not consistent F_d), it can also apply nipping and pruning in both directions.

In addition, BiMax-BS$^*_F$ performs F*-leveling* (instead of the cardinality criterion for each node): The search continues in its current direction d until no open node with the current $fmin_d$ is left, then $fmin_d$ is increased. In order to make good use of the *Max* method, the differences in one frontier used for improving the heuristic values in the other should be as high as possible. Therefore, it is useful to have reached the next level of f-values for improving the search in the other direction. That is why BiMax-BS$^*_F$ does not change its search direction for each node. Whenever the next level of f-values is reached in the current search direction, it decides about the new search direction using the cardinality criterion. In this way, statement number 5 in procedure *GenericBS** is instantiated.

More algorithms and details as well as admissibility proofs are documented (in German) in the diploma thesis of the first author of this paper. The major difficulty in developing BiMax-BS$^*_F$ was that normally an improved heuristic is utilized in a best-first search for sorting the open nodes. This turned out, however, as less useful here than its utilization for pruning nodes, which is more usual in depth-first search.

4 EXPERIMENTAL RESULTS

Now let us have a look on specific experimental results for finding optimal solutions to the set of 100 Fifteen Puzzle problems. We compare our new algorithms with those that achieve the best published results in this domain. We also compare them with IDA* as the linear-space reference algorithm, as well as A* and BS* as the unidirectional and bidirectional best-first search references. All the compared algorithms used no domain-specific knowledge about the puzzle other than the Manhattan distance heuristic. The processor used was an AMD K7-1800+ and the main storage available was up to 2 Gbytes.

A* stored up to 43 million nodes. On the set of 78 problems that it was able to solve with this memory, our best algorithm BiMax-BS$^*_F$ just generated 14.0% of the nodes generated by A* and needed 15.9% of its time.

On the 95 problems solvable by BS*, BiMax-BS$^*_F$ just generated 26.0% of the nodes and needed 29.3% of the time of BS*. Using the *Max* heuristic in one direction only for sorting the nodes in the OPEN list was not that effective, but our new algorithm Max-BS* also generated just 38.3% of the nodes and needed 40.9% of the time of BS*. Max-Switch-A* [4] has about the same results, but BiMax-BS$^*_F$ has clearly the best results, due to its more effective use of the *Max* heuristic in both directions.

Fig. 3 shows a comparison of several algorithms (again in terms of the average number of node generations and their running times) for solving the complete set of 100 problem instances. The data are again normalized to the respective search effort of IDA*. Just to give an idea of the overall difficulty of the given problem set, note that IDA* generates some 363 million nodes on average, which needs 36.11 seconds on the given machine to find an optimal solution to one problem instance. For the single most difficult problem, IDA* needs 628.9 seconds to find an optimal solution, while BiMax-BS$^*_F$ just uses 68.5 seconds.

The algorithm Trans [12] implements a form of enhanced iterative-deepening search that uses a transposition table for IDA*. Trans utilizes its table actually for two purposes, for recognizing transpositions and for caching the best heuristic values acquired dynamically. Fig. 3 shows the best result that we achieved with this algorithm, where it used a hash table with 6 million nodes. In fact, this table just uses an order of magnitude less storage than the 2 GBytes available. While the number of nodes generated slowly but steadily decreased with increasing table size, the running time became worse with larger tables. One reason is the longer initialization time of the table in the beginning, which is measurable for the very easy problems. The other reason is that a larger table means hashing in a higher number of nodes. In an efficient implementation of the Fifteen Puzzle such as the one used in our experiment, even the effort of hashing causes an overhead that cannot be ignored. Compared to a previously published result in [3], the overall overhead of memory access is relatively larger. We think that this is due to the CPU being relatively faster than memory access on the new machine.

Another technique to prune duplicate nodes was proposed in [13], using a finite state machine. Its results are not included in Fig. 3, since we lack data on the running time (no such data are given in [13], and we did not re-implement this technique). IDA* employing

this pruning technique generated 100.7 million nodes on the same set of instances as reported in [13], which means 27.7 percent of the number of nodes generated by pure IDA*. Ignoring that the finite state machine must be built in a pre-processing stage first, we estimate that this approach would require about 30 percent of the running time of IDA*. Since the best (bidirectional) algorithms are clearly faster than this approach, however, it is not necessary to have the exact data for the comparison available.

In principle, we have provided all the available storage to BIDA* [10], the most efficient algorithm of the perimeter approach. But it cannot make use of it. Even to the contrary, a comparison of results with perimeter depths 7, 14 and 16 as shown in Fig. 3 suggests that the version with the least memory use (7) achieved the best running time, possibly since it fit into the cache. We actually determined this optimum for the given machine empirically by letting it run with many different perimeter depths. In effect, we tuned this competitor to achieve its best result under the given conditions. The version with perimeter depth 14 was the one reported as best originally in [10], and the one with 16 in [3].

Also Max-BAI-Trans cannot really utilize the whole memory available. The version (4,1) stores 4 million nodes for the *Max* heuristic and 1 million nodes for the Trans part. This version just needs 256 MBytes. We also tuned this competitor and achieved its best results with 4 million nodes for the *Max* heuristic and 8 million nodes for the Trans part. Increasing the memory size in either way resulted in longer running times. The explanation for the different result in [3] as compared to IDA* is the same as for Trans above.

Our new algorithm BiMax-BS$^*_F$ can store 43 million nodes in the given 2 GBytes of memory. It can utilize it well by achieving the best overall running time. (Even the number of generated nodes is comparable to BIDA*, which performs "front-to-front" evaluations that are much more expensive per node searched.) Its (constant) overhead per node results in a factor of about 12 compared to the running time of IDA* on the given machine, which runs in cache throughout and does not access a hash table. But the savings in terms of nodes generated overcompensate this to a great extent.

The superiority of BiMax-BS$^*_F$ in terms of running time over its best competitors is statistically significant. For example, the probability that the improvement of the running time over BIDA* (with perimeter depth 7) is due to chance fluctuation is 0.47 percent according to a test that compares the means of the paired samples of the absolute running times. Compared to the best version of Max-BAI-Trans (4,8) on the machine used, it is 2.94 percent. Note, that we compare BiMax-BS$^*_F$ here with versions of its competitors that were tuned for their advantage.

Let us also briefly make a comparison relative to the difficulty of the problem instances. BiMax-BS$^*_F$ is particularly better on the more difficult problems, while it is also often worse on the easy problems. Since these can be solved in the order of a second, this does not matter much. It may matter, however, whether to wait some 20 seconds (or up to 68 seconds), or twice as long (or even longer) for the solution to a more difficult problem (the most difficult one).

5 CONCLUSION

Much larger main storage is available today than previously. Depth-first search cannot utilize it, but additionally available memory may be given to some memory-bounded search algorithm. However, as indicated by their results above, memory-bounded algorithms do not necessarily scale up with increasing memory size. In addition, some of the better algorithms for memory-bounded search need to be tuned for a given machine using their parameters, whose values may have to be determined empirically first.

Using the standard benchmark of Fifteen Puzzle problems for a case study, this paper shows that traditional best-first search can find optimal solutions now (even with the old Manhattan distance heuristic). It solves such problems faster overall, especially the more difficult ones. In order to achieve this result, algorithmic improvements were necessary for utilizing a dynamically improved heuristic. We had to overcome several problems, especially how to make good use of the dynamic *Max* heuristic in a BS*-like algorithm.

This paper also shows that *bidirectional* heuristic search was necessary for this result, since no known *unidirectional* best-first search can solve all the given problem instances (A* being known as optimal in terms of expanded nodes). Also in contrast to switching to unidirectional search, we show now that the continuation of bidirectional search works better with the dynamic *Max* heuristic. It is important to be able to utilize it in *both* directions.

We cannot make any claims yet about the performance of this particular approach on other problem domains. Still, the important lesson learned is that traditional best-first search is able to solve problems that it was believed to be incapable of. This search paradigm is out of question for solving very difficult problems or when having much time available for finding solutions. For solving problems in "real-time" situations, however, where humans are waiting for a solution in the order of seconds to minutes, it can make an important difference. On currently available machines, traditional best-first search can solve relatively difficult problems efficiently.

REFERENCES

[1] J. Culberson and J. Schaeffer, 'Searching with pattern databases', in *Advances in Artificial Intelligence*, ed., G. McCalla, 402–416, Springer-Verlag, Berlin, (1996).

[2] R. Dechter and J. Pearl, 'Generalized best-first strategies and the optimality of A^{*}', *J. ACM*, **32**(3), 505–536, (1985).

[3] H. Kaindl and G. Kainz, 'Bidirectional heuristic search reconsidered', *Journal of Artificial Intelligence Research (JAIR)*, **7**, 283–317, (1997).

[4] H. Kaindl, G. Kainz, R. Steiner, A. Auer, and K. Radda, 'Switching from bidirectional to unidirectional search', in *Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*. San Francisco, CA: Morgan Kaufmann Publishers, (1999).

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

[6] R.E. Korf and A. Felner, 'Disjoint pattern database heuristics', *Artificial Intelligence*, **134**, 9–22, (2002).

[7] R.E. Korf and L.A. Taylor, 'Finding optimal solutions to the Twenty-Four Puzzle', in *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 1202–1207. Menlo Park, CA: AAAI Press / The MIT Press, (1996).

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

[9] E.L. Lawler and D. Wood, 'Branch-and-bound methods: a survey', *Operations Research*, **14**(4), 699–719, (1966).

[10] G. Manzini, 'BIDA*: an improved perimeter search algorithm', *Artificial Intelligence*, **75**(2), 347–360, (1995).

[11] I. Pohl, 'Bi-directional search', in *Machine Intelligence 6*, pp. 127–140, Edinburgh, (1971). Edinburgh University Press.

[12] A. Reinefeld and T.A. Marsland, 'Enhanced iterative-deepening search', *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, **16**(12), 701–709, (July 1994).

[13] L.A. Taylor and R.E. Korf, 'Pruning duplicate nodes in depth-first search', in *Proc. Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 756–761. Menlo Park, CA: AAAI Press / The MIT Press, (1993).

[14] W. Zhang and R.E. Korf, 'Depth-first vs. best-first search: new results', in *Proc. Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 769–775. Menlo Park, CA: AAAI Press / The MIT Press, (1993).