How to Use Limited Memory in Heuristic Search

Hermann Kaindl
Siemens AG Osterreich
Geusaugasse 17
A 1030 Wien, Austria
kaih @siemens.co.at

Gerhard Kainz
Gemeindeberggasse 14
A-1130 Wien
Austria

Angelika Leeb
MIT Lab. f. Comp. Sc.
545 Technology Square
Cambridge, MA 02139
U.S.A.

Harald Smetana
Lerchengasse 12
A 3430 Tulln
Austria

Abstract

Traditional *best-first* search for optimal solutions quickly runs out of space even for problem instances of moderate size, and *linear-space* search has unnecessarily long running times since it cannot make use of available memory. For using available memory effectively, we developed a new generic approach to heuristie search. It integrates various strategies and includes ideas from *bidirectional* search. Due to insights into different utilizations *of* available memory, it allows the search to use *limited* memory effectively. Instantiations of this approach for two different benchmark domains showed excellent results that are statistically significant improvements over previously reported results: for finding optimal solutions in the 15-Puzzle we achieved the fastest searches of all those using the Manhattan distance heuristic as the only knowledge source, and for a scheduling domain our approach can solve much more difficult problems than the best competitor. The most important lessons we learned from the experiments are first, that also in domains with symmetric graph topology selecting the right search direction can be very important, and second, that memory can—under certain conditions—be used much more effectively than by traditional best-first search.

1 Introduction

Best-first search in the tradition of *A** [Hart *et* a/., 1968] typically requires exponential space. Therefore, it quickly runs out of space even for problem instances of moderate size when searching for optimal solutions.

In contrast, *linear-space* search like *IDA** [Korf, 1985], *RBFS* [Korf, 1993] and *DFBB* [Lawl er and Wood, 1966] does not suffer from memory limitations. Typically, there is even much more memory available than needed by such algorithms. Since they cannot make use of such memory, however, their running time is unnecessarily long.

One of the major problems of heuristic search is how to use available but limited memory effectively. Pure *unidirectional* approaches to utilizing limited memory still did not lead to convincing results [Chakrabarti *et al,* 1989; Ghosh *et ai,* 1994; Reinefeld and Marsland, 1994; Russell, 1992; Sen and Bagchi, 1989]. Therefore, we propose to consider in addition ideas from *bidirectional* search [Kaindl and Khorsand, 1994; Koll and Kaindl, 1993; Kwa, 1989; Pohl, 1971].

In this paper we integrate various unidirectional strategies and include ideas from bidirectional search in a generic approach to heuristic search. Due to insights into different utilizations of available memory, our approach allows the search to use *limited* memory effectively.

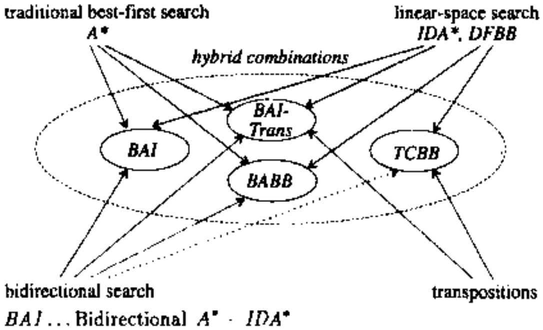
First, we present our new generic approach to heuristic search that specifically focuses on using limited memory. Then we show how this approach can be appropriately instantiated for two very different domains with few and many distinct cost values, respectively. For both domains, we report experimental data that represent significant improvements over previously published results. Finally, we compare our approach more generally with related work.

2 A Generic Approach to Heuristic Search Using Limited Memory

We developed a new generic approach to heuristic search that integrates various approaches and typically leads to hybrid combinations of search algorithms. One of the main ideas to address the memory problem is to combine linear-space search with conventional best-first search in a bidirectional style. Fig. 1 gives an overview and indicates how our new algorithms integrate ideas from various approaches in hybrid combinations—this will be discussed in detail below. First, we explain the *generic approach* generally. Its major steps are:

1. Assign the search directions and the available memory to the traditional best-first and the linear-space algorithm, respectively.
2. Perform traditional best-first search using some or even nearly all of the available memory.
3. Unless the best-first search has already found an optimal solution, use a linear-space algorithm in the reverse direction. Use the memory structure built up by the previous best-first search, possibly together with additional memory that is still available.

Note, that the linear-space algorithm per se only requires little memory, but it can utilize additional mem-



BAI ... Bidirectional *A** - *IDA**
BAI-Trans ... Bidirectional *A** - *IDA** - *Trans*
BABB ... Bidirectional *A** - *depth-first Branch-and-Bound*
TCBB ... *Transpose-and-Cache* - *depth-first Branch-and-Bound*

Figure 1: Overview of the generic approach and some instantiations.

ory tnrougn direct, access, e.g., via nasning.

Since this approach does not allow for changing the search direction more than once, it can be viewed as a non-traditional form of bidirectional search. In particular, the recently proposed perimeter search [Dillenburg and Nelson, 1994] fits into it. However, we explicitly do not propose to use a wave-shaping strategy1 (see also [do Champeaux, 1983]), since it is inefficient in terms of running time.

Under certain circumstances to be discussed below, one or more of the steps can also be omitted during instantiation. For instance, if the domain shows a strong asymmetry, the search direction is known before the search begins. Moreover, our generic approach can also be instantiated as unidirectional search.

Since this generic approach has to be instantiated for appropriate use, we must give guidelines for doing so. First of all, it is important to understand how given memory can be utilized during heuristic search. While some of these utilizations appear to be well understood, we could not find a clear categorization in the literature. Note that item 5(b) given below appears to have been previously overlooked. We distinguish the following utilizations of given memory:

1. for ordering the sequence of node generations:
   traditional best-first search like A* organizes the memory as a priority queue for this purpose, while more recent linear-space RBFS has to backtrack;
2. for storing state information of generated nodes:
   for instance, in the well-known 15-Puzzle the configuration of tile positions can be stored;
3. for finding transpositions" in a directed acyclic graph in order to avoid "treeification" during search: again, A* is a well-known example, while IDA* and DFBB normally cannot recognize transpositions without extra memory (which may also be used as in [Taylor an d Korf, 1993]);
4. for caching information about heuristic estimates: three cases can be distinguished here:
   (a) static heuristic value:
   storing such values avoids recomputation;
   (b) more accurate information closer to a goal:
   memory-bounded algorithms like MREC [Sen and Bagchi, 1989] propagate better heuristic estimates found during their linear-space searches back to the stored part of the search graph;
   (c) heuristic information computed between two search frontiers:
   wave-shaping approaches to bidirectional search utilize memory to store such values computed during the search;
5. for finding solutions:
   two cases can be distinguished here:
   (a) finding solutions at all:
   traditional bidirectional search like BHPA [Pohl, 1971] and IBS* [K611 and Kaindl, 1993] needs the memory to find solutions by recognizing meetings of the search frontiers;
   (b) finding solutions earlier:
   the non-traditional approach to bidirectional search described below uses memory to find solutions earlier than without the memory.
6. for proving solution quality:
   bidirectional heuristic search utilizes given memory for storing estimates of the optimal solution cost to be used for proving the quality of a solution found.

1 Wave-shaping has to compute heuristic estimates between all nodes in one search frontier and all nodes in the other, i.e., the effort is proportional to the cross product of the numbers of nodes in the frontiers.

2Transpositions arise when different paths lead to the same node.

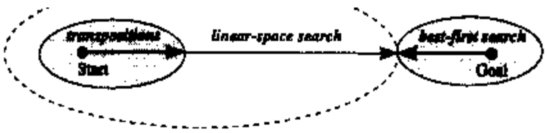


Figure 2: An approach to using memory on both sides.

The combination of search ideas to be chosen during an instantiation of our generic approach should utilize the given memory in several different ways. This typically leads to higher efficiency since their advantages are more or less disjoint and nearly add up, while using the same amount of storage just for one purpose may not give the same pay-offT. Fig. 2 shows a useful specialization of our generic approach that uses memory on both sides of the search space: a transposition table [Reinefeld and Marsland, 1994] on one side, and the memory of a traditional best-first search on the other. The former is used for finding transpositions and caching more accurate heuristic evaluations closer to the goal. The latter first of all orders the sequence of node generations and helps finding transpositions in another part of the search space, and finally it supports finding solutions earlier.

Of course, an instantiation should make use of any domain-specific information available. In particular, it should combine those unidirectional search algorithms that best suit the properties of the domain (see, e.g., [Rao *et al,* 1991; Zhang and Korf, 1993]). For instance, in some domains *IDA** is the choice, while in others *DFBB* is much better.

3 Instantiating for a Domain with Few Distinct Cost Values: 15-Puzzle

First we show how our generic approach can be instantiated for a domain that is characterized by having only few distinct cost values: the well-known 15-Puzzle. Under this condition, it is reasonable to select *IDA** for the linear-space search part.

Moreover, we can assume to have a *monotone* evaluation function: the Manhattan distance. Since *A** makes good use of monotone heuristics [Dechter and Pearl, 1985], we select it for the part of the best-first search.

Based on the key idea of bidirectional search, we let A* and *IDA** search in opposite directions in steps 2 and 3 of our generic approach, respectively. Fig. 1 illustrates this instantiation that leads to *BAI* (Bidirectional *A* - IDA*).*

According to the idea of using memory in various ways, we may also give the *IDA** search some part of the available memory as a transposition table. Fig. 2 illustrates this approach generally. We call this variant of *BAI* due to the use of this table *BAI-Trans* (see also Fig. 1).

If *A** cannot find a solution using the given memory, then *IDA** searches in the reverse direction towards the frontier *of* the prior search. Since we consider the case of finding *optimal* solutions, this search cannot always terminate immediately after a solution is found. A better solution may exist, and the algorithm must find an optimal one and subsequently prove that it is optimal.

More technically, the *IDA** part must be changed slightly. Instead of having to find the goal node, a solution is found whenever the depth-first search meets the frontier of the opposing *A** search. If the cost of this solution is smaller than the cost of the best solution found so far (or if it is the first solution found) then its value is stored. Of course, the cost of the best solution found so far may be sub-optimal, or the algorithm does not yet know that, it is already optimal. However, if the stored value does not exceed the non-overestimating threshold of the *IDA** part, then its depth-first search is exited successfully with an optimal solution.

In addition to these necessary changes, the *IDA** part has the advantage to start with an increased initial threshold based on an *admissible* estimate of the optimal solution cost as determined by the *A** part. Since we assume a monotone heuristic *h,* the minimum of $f = g + h$ for all nodes in OPEN is always an admissible estimate. Therefore, if this estimate is higher than the usual initial threshold of *IDA**, then it can be used here instead.

Moreover, it is not necessary to have the *IDA** part search again in the space already explored by *A*.* More technically, when the depth-first search invoked by *IDA** meets a CLOSED node of the opposing *A** search frontier, this branch can be cut off (meeting an OPEN node is in general insufficient). We call this *nipping* according to an analogous method described in [Kwa, 1989].

In an efficient implementation of the 15-Puzzle even the effort of hashing at every node causes an overhead that cannot be ignored. Therefore, we implemented *BAI* in such a way that it avoids hashing at those nodes where—based on the heuristic estimate—it knows that the frontier of the opposing A* search is yet out of reach.

According to step 1 of our generic approach, the search directions must be assigned to the *A** and the *IDA** part, respectively. For traditional bidirectional search, Pohl [Pohl, 1971] proposed and used a *cardinality criterion* for the problem of determining the frontier from which to select a node for expansion: continue searching from the frontier with fewer OPEN nodes. While this is utilized for each node expansion in traditional bidirectional search algorithms, *BAI has* to decide this issue once at the very beginning of the whole search. When the search space is sufficiently symmetric, the initial search direction can be determined at random. When the search space is at least slightly asymmetric and no specific knowledge for determining the search direction is available, it seems reasonable to make shallow probes into the search space from both sides and to use the idea of the cardinality criterion. Since *BAI* incorporates *IDA*,* using this algorithm also for *probing* is consistent with the overall approach. For instance in the 15-Puzzle, the first few iterations of *IDA** are searched from both sides, and the direction with fewer generated nodes is assigned to the *IDA** part of the overall search, since especially for difficult problems it will have to search much deeper than the *A** part.

Let us shortly discuss the behavior of *BAI.* In the *best case,* it would seem to be the same as *A*.* In fact, *BAI* can even be better than pure *A*. BAI* assigns the search direction dynamically, which can lead to better results than systematically going in one direction. In the *worst case, BAI* has to perform the part of *A**, without savings in the *IDA** part (except the effect of nipping).

A key question is how *BAI* saves effort without having enough memory available for completing the *A** search. Primarily, it can save one or more of *IDA*'s* iterations. Due to the better initial threshold, some of the early iterations can be saved. Since the earlier iterations are comparably cheap, this helps much less than saving the last iteration. The search can also be terminated *after a complete* iteration *of IDA** if the cost of the best solution already found is not larger than the new increased threshold. Therefore, large savings are possible when *BAI* terminates earlier than pure *IDA*.*

4 Instantiating for a Domain with Many Distinct Cost Values: Single Machine Scheduling

Now let us show how our generic approach can be instantiated for a domain that is characterized by having *many* distinct cost values: a *scheduling domain* described and used for experiments in [Sen and Bagchi, 1993; Townsend, 1978].[3]

[3] Since this benchmark domain is not widely known, we sketch it shortly. It deals with one-machine job sequence problems of the following form. Jobs *J,* with processing times

For problems with many distinct cost values, *IDA** is known to be much less efficient. But it is reasonable to select *DFBB* for the linear-space search part, assuming that a good upper bound can be determined quickly—which is the case in this kind of problem.

Again, a monotone evaluation function is available, so we can analogously to *BAJ* select *A** for the part of the best-first search. Based on the key idea of bidirectional search, we let *A** and *DFBB* search in opposite directions in steps 2 and 3 of our generic approach, respectively. Fig. 1 illustrates this instantiation that leads to *BABB* (Bidirectional *A* depth-first Branch-and-Bound).* Technically, the computation of the heuristic is different in the backward direction, and the necessary modification is not completely trivial. While in the forward direction all the jobs on the current path are already included, in the backward direction the jobs on the path have *not* yet been included. The best approach we found is to define a new problem that excludes all these jobs, and to compute the heuristic for this problem.

Due to the above mentioned asymmetry of the arc cost distribution, the better search direction can be determined a priori in this domain. (The reasons are discussed below.) Therefore, step 1 of our generic approach can be omitted here.

For the same reason, another possibility to instantiate our generic approach here is to omit also the best-first search part completely and to provide all the available memory in the form of a transposition table. Analogously to its use in *IDA*,* it both finds transpositions and caches dynamically acquired heuristic information. Therefore, we call this algorithm *TCBB* (*Transpose-arid-Cache - depth-first Branch-and-Bound).* Still, the bidirectional idea of starting on either side of the space is important here, which is illustrated by the dotted arrow in Fig. 1.

An obvious advantage of both *BABB* and *TCBB* over *A** is their ability to continue searching although the memory is too small to store all the nodes. A much less obvious advantage of *TCBB* over *A** is that—under certain conditions—*TCBB* utilizes available memory much more effectively. This will be explained together with empirical results below.

5 Experimental Results

In our experiments, we compared *BAI, BAI-Trans, BABB* and *TCBB* with other algorithms on the task of finding optimal solutions in two different domains. From the derivations of our algorithms, it should be intuitively clear that these algorithms are *admissible,* i.e., if a solution exists, they terminate with an *optimal* solution. Formal proofs can be found in [Kainz, 1995].

a, and penalty constants p, (associated with completing a job at time *ti)* are submitted to a one-machine job-shop. *ti* is the sum of the times *aj* of all jobs *j* on the currently evaluated path. The penalty function is *Gi(t,) = pxt21.* All the jobs must be sequenced on the machine in such a way that the sum of all penalties is minimized. Important properties of this domain are a symmetric graph topology and an asymmetric distribution of arc costs that is due to the quadratic penalty function.

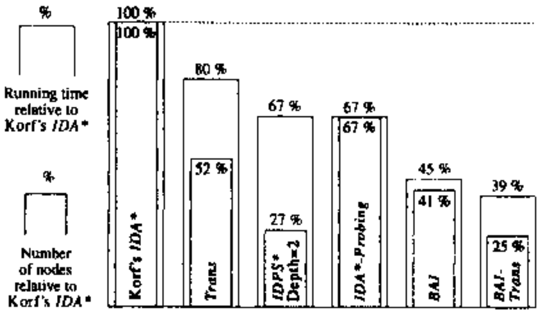


Figure 3: Comparison on the 15-Puzzle (100 instances).

5.1 15-Puzzle

Now let us have a look on specific experimental results for finding optimal solutions to a set of (sliding-tile) 15-Puzzle problems.4 We compare our new algorithms *BAI* and *BAI-Trans* on these problems with other algorithms that achieve the best results known here yet without using domain-specific knowledge about the puzzle other than the Manhattan distance heuristic: *IDA*, Trans* [Reinefeld and Marsland, 1994] and *IDPS** (perimeter search) [Dillenburg and Nelson, 1994]. *RBFS has* the potential to be better than *IDA** on the sliding-tile puzzle, but actually its results are slightly worse on this specific problem set according to [Korf, 1993] due to noise in the tie-breaking on the last iteration.

Fig. 3 shows a comparison of these algorithms in terms of the average number of node generations and their running times. The data are normalized to the respective search effort of *IDA*. Trans* (using 256k nodes of memory) achieves savings of nearly half of the nodes compared to *IDA*,* and although the effort for hashing slows it down, *Trans* is faster than *IDA** since it utilizes its memory well.6 *IDPS** saves even 73 percent of the nodes

4 We used the complete set of 100 instances from [Korf, 1985].

5In an efficient implementation of the 15-Puzzle, even checking of whether a node generated by *IDA** is in the opposing search frontier means a measurable overhead, although this can be done efficiently via hashing. The reason is that node generation and evaluation can be done very efficiently for the sliding-tile puzzles [Korf, 1993]. Even the machine architecture can influence the relative running time up to a certain degree. Therefore, we note that our data were gained using a Convex C3220. On this machine, *IDA** searches 210k nodes per second.

6 The data in the figure were gained using a re-implementation of *Trans* based on efficient code provided by Jonathan Shaeffer. Note the different way of presenting the results: *absolute* data in our figure vs. *relative* to problem difficulty in [Reinefeld and Marsland, 1994]. We had to re-implement *Trans,* since the detailed data in terms of node generation numbers provided by Alexander Reinefeld and the data reported in [Reinefeld and Marsland, 1994] were insufficient to get comparable data on the running times. Actually, *Trans+Move* is the best algorithm described in [Reinefeld and Marsland, 1994], but its *absolute* results are less than one percent better than those of *Trans.* Therefore, we did

generated by *IDA**, but due to its wave-shaping overhead only 33 percent of the running time.7

The results of our new algorithms shown in Fig. 3 were generated by giving them also 256k nodes of memory, and determining the search direction via probing through *IDA** searching its first three iterations (this generates only very few nodes but already indicates the better search direction with a relatively high reliability of 80 percent). *BAIsaves* more nodes than *Trans,* and it is also faster. The overhead through hashing is smaller due to our strategy of avoiding it if possible. Our combination *BAI-Trans* saves three quarters of *IDA*ys* nodes and 61 percent of its running time. This shows that the respective advantages of using memory in these different ways nearly add up, although the available memory for each of its parts is just half of the memory given to each *BAI* and *Trans,* i.e., *BAI-Trans* altogether has 256k nodes of memory available here. Although *BAI-Trans* does not use wave shaping, it generates slightly fewer nodes than *IDPS** with perimeter depth 2. Because of avoiding the overhead of wave shaping, *BAI-Trans* is clearly faster than *IDPS** (although with this perimeter depth this overhead is relatively very small). In order to make sure that these results are not due to chance fluctuations, we performed statistic tests.8 The superiority of *BAI- Trans* in terms of running time over pure *IDA**, pure Trans and *IDPS** as shown in the figure is statistically significant.

Although the search space of the sliding-tile puzzle appears to be quite symmetric, it is interesting to see how much can be gained here just by selecting the search direction dynamically. Therefore, we conducted a special experiment with a variant of *IDA** that just uses our approach for selecting the search direction: *IDA*-Probing.* Fig. 3 shows that even this simple linear-space algorithm is slightly faster than *Trans*, since it has no overhead in running time. Moreover, the advantage of *IDA*-Probing* over pure *IDA** is statistically significant. In order to see how well probing via three iterations already indicates the better search direction, we compared its result with that of a perfect oracle. Using it would still generate 64 percent of *IDA*'s* nodes, i.e., *IDA*-Probing* with an overhead in generated nodes for determining the

not re-implement *Trans+ Move a.nd* cannot include it into the figure, lacking data on the running times.

7The results reported in [Dillenburg and Nelson, 1994] are based on runs using a different sample set of the 15-Puzzle, and a different perimeter depth. Using the same perimeter depth (4), the results on Korf's set with our re-implementation are even better in terms of the number of node generations, but very much slower in terms of running time (even slower than *IDA*).* In personal communication with John Dillenburg it turned out that their implementation of *IDA** is slower than Korf's one that we are using by a factor of about 60 per generated node. In such an implementation the overhead especially of wave shaping does not show up that clearly as it does in an efficient one. Since smaller perimeter depth means fewer stored nodes and therefore less overhead through wave shaping, the perimeter depth 2 results in better running time, and consequently we show these data in our figure.

8 More details on the statistic tests used can be found in [Kaindl *et al.*, 1994; Kaindl and Smetana, 1994].

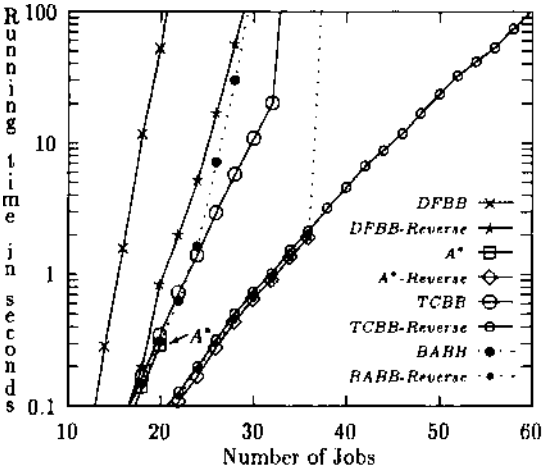


Figure 4: Comparison on the Scheduling Problems.

search direction of only less than 0.1 percent is overall just 3 percent worse than this. Systematically searching in the backward direction, however, is not significantly better due to high standard deviations, although it saves 17 percent over systematically searching in the forward direction.

In summary, instantiating our generic approach as described lead to the fastest searches for finding optimal solutions on the 15-Puzzle of all those using the Manhattan distance heuristic as the only knowledge source. While saving 61 percent of the running time of pure *IDA** may not seem impressive, the difficulty *of* significantly improving its results may be appreciated when looking at the many less successful attempts with various algorithms (see the section on related work below).

5.2 Single Machine Scheduling

The best results yet reported for the scheduling problem that we chose for our experiments are those of *A** that searches systematically in the forward direction: finding optimal solutions to random problems of 28 jobs in an average of some few seconds each with less than 300k nodes of memory. As compared in [Sen and Bagchi, 1993], these results signified a very strong improvement over previous approaches that did not utilize the graph structure of the search space.

Fig. 4 shows *A** data and the results of *DFBB* in comparison to the results that we achieved through instantiating our generic approach appropriately. Except *DFBB* and *DFBB-Reverse,* all the algorithms were given the same amount of memory: 16k nodes.9 The data point

9 We chose this amount of memory for showing the high efficiency of *TCBB* in regard to memory utilization. We used the same 100 randomly generated problem instances per "number of jobs" as [Sen and Bagchi, 1993], since Anup Sen made the generator available to us. The figure shows the results in terms of running time, but the numbers of generated nodes are quite the same, since computing the heuristic values dominates the running time. Note, that the data are shown on a logarithmic scale.

annotated in the figure via *"A*"* indicates the result with the previously best method. The main discovery was that systematically searching in the reverse direction—from the goal to the start—yields much better results: see the data of *DFBB-Reverse* and *A*-Reverse.* These results are very surprising since the graph topology of the search space is symmetric. According to current theory, such a strong effect of reversing the search direction would have been attributed to a strong asymmetry of the graph topology.

Actually, the reason for this phenomenon is due to the asymmetric *distribution of arc costs* induced by the quadratic penalty function in these problems. In order to find this reason, we made special experiments, and since this phenomenon also occurs without using heuristic values, it is at least not primarily related to their accuracy. We can explain it as follows: when searching in the forward direction, many nodes must be generated in order to find that a path is bad since the arc costs are initially small; in the backward direction, however, the arc costs are initially large and vary strongly, which allows the identification of bad paths already after relatively few node generations.

Independently of search direction, *A** still runs out of space with increasing problem difficulty, and pure *DFBB* cannot recognize transpositions and has excessive running times due to the huge number of nodes searched. Our algorithm *BABE* is better than *A** insofar as it can continue searching when *A** already has to quit due to lack of memory on difficult problems with many jobs. However, it does not utilize the given resources optimally.

Although *A** is slightly faster than *TCBB* due to generating slightly fewer nodes, our new algorithm *TCBB* utilizes its memory much better than *A** (and *BABB).* While A* runs out of 16k nodes of memory (in the forward direction) even for problems with more than 20 jobs, *TCBB* needs no more memory for problems of 32 jobs. Even when it cannot store all of the searched nodes on more difficult problems, it still finds solutions.

One reason for the better utilization *of* memory by *TCBB* compared to *A** in this domain is that it is possible here for *TCBB* to find a good upper bound quickly that avoids storing all the nodes with a higher estimated cost. In contrast, *A** is unaware of an upper bound and stores many nodes with bad values. Another reason is that in these problems the branching degree is relatively high, and *TCBB* can avoid storing many nodes, which reinforces the effect above. In contrast, *A** stores all of them due to its strategy of generating and storing all the successors at once at node expansion.

Due to the strong asymmetry of the arc cost distribution, this unidirectional search algorithm is even much better in the reverse direction: *TCBB-Reverse*—but the bidirectional view was necessary to discover the strong asymmetry in this domain. *TCBB-Reverse* can solve even problems with 60 jobs in an average of about 100 seconds, which shows how well our approach scales up. While *A*-Reverse* is much better than *A** in the forward direction, it still runs out of 16k nodes of memory for problems with more than 36 jobs.

In summary, we achieved very strong overall improvements with our approach in this scheduling domain compared to the best results reported in the literature. In particular, our approach can solve much more difficult problems with the same amount of memory and within reasonable time.

6 Related work

Originally, bidirectional heuristic search did not work as expected [de Champeaux, 1983; Kwa, 1989; Pohl, 1971]. Recent results show that bidirectional search has the potential to improve on unidirectional search [Kaindl and Khorsand, 1994; Koll and Kaindl, 1993]. Unfortunately, traditional bidirectional search requires rather complicated mechanisms that make it difficult to implement. Moreover, in domains that can be implemented with fast node generation and computation of the heuristic function like the 15-Puzzle, these mechanisms imply a certain overhead. Therefore, the generic approach in this paper tries to utilize key ideas of bidirectional search in an efficient manner.

A bidirectional algorithm sketched in [Korf, 1985] employs *DFID (depth-first iterative-deepening* without using heuristic knowledge). Since its space requirement is still $0(6^{d}/^{2})$, it cannot solve difficult problems.

Perimeter search [Dillenburg and Nelson, 1994] is a non-traditional approach to bidirectional search that may look very similar to our algorithm *BA1.* However, the key difference is the use of a form of wave shaping in perimeter search, that makes it inefficient in terms of running time.

Apart from bidirectional search there are some relations to *unidirectional* search algorithms with reduced space requirements: *MREC* [Sen and Bagchi, 1989], *MA** [Chakrabarti *et al.,* 1989], *SMA** [Russell, 1992], *ITS* [Ghosh *et* a/., 1994], and the approach of using certain tables for *IDA** [Reinefeld and Marsland, 1994]. From these, *Trans* and *Trans-\-Move* [Reinefeld and Marsland, 1994] gave the best results on the 15-Puzzle in terms of running time that we are aware of. Similarly to some of this referenced work, our *BAI* algorithm can be viewed as saving nodes otherwise searched by *IDA*.* Our new algorithm *TCBB* is clearly analogous to *Trans,* but due to including *DFBB* instead of *IDA** it is much more efficient in domains with many distinct cost values and the possibility of getting reasonable upper bounds on the solution cost.

7 Conclusion

Our approach makes use *of* some known ideas and algorithms. However, this paper contains several new ideas and results:

- We developed a generic approach to heuristic search that utilizes limited memory effectively.
- We categorize several different utilizations of given memory for heuristic search and identify a new one in our approach.
- During experiments with instantiations of our approach in two different domains, we learned that even for nearly symmetric graph topology, selecting the search direction can be important. Especially the distribution of arc costs can be crucial.

- We found that—under certain conditions—available memory can be utilized much more effectively than by *A*.*
- In both selected domains—the 15-Puzzle and a special scheduling domain—we achieved significantly better results with our approach than those previously reported in the literature.

In summary, our new generic approach to heuristic search integrates various strategies and includes ideas from bidirectional search. Due to insights into different utilizations of available memory, it allows the search to utilize *limited* memory effectively.

Acknowledgments

Our implementations are based on the very efficient code of *IDA** and *A** for the puzzle made available by Richard Korf and an efficient hashing schema by Jonathan Shaeffer, as well as on code of *A** and *DFBB* for the scheduling domain by Anup Sen. Alexander Reinefeld and Tony Marsland provided the data on the generated nodes for *Trans* and *Trans+Move* as well as comments on an earlier draft of this paper. Finally, we acknowledge the useful comments on an earlier draft by Wilhelm Barth, Wolfgang Schmid and Anup Sen.

References

[Chakrabarti *et ai,* 1989] P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. DeSarkar. Heuristic search in restricted memory. *Artificial Intelligence,* 41 (2): 197-221, 1989.

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

[Dechter and Pearl, 1985] R. Dechter and J. Pearl. Generalized best-first strategies and the optimality of *A*. J. ACM,* 32(3):505-536, 1985.

[Dillenburg and Nelson, 1994] J.F. Dillenburg and P.C. Nelson. Perimeter search. *Artificial Intelligence,* 65:165-178, 1994.

[Ghosh *et ai,* 1994] S. Ghosh, A. Mahanti, and D.S. Nau. ITS: an efficient limited-memory heuristic tree search algorithm. In *Proc. AAAI-94,* pages 1353-1358, 1994.

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

[Kaindl and Khorsand, 1994] H. Kaindl and A. Khorsand. Memory-bounded bidirectional search. In *Proc. AAAI-94,* pages 1359-1364, 1994.

[Kaindl and Smetana, 1994] H. Kaindl and H. Smetana. Experimental comparison of heuristic search algorithms. In *AAAI-94 Workshop on Experimental Evaluation of Reasoning and Search Methods,* pages 11-14, 1994.

[Kaindl *et ai,* 1994] H. Kaindl, A. Leeb, and H. Smetana. Improvements on linear-space search algorithms. In *Proc. ECAI-94,* pages 155-159, 1994.

[Kainz, 1995] G. Kainz. Heuristische Suche mit begrenztem Speicherbedarf. Doctoral dissertation, Technische Universitat Wien, 1995. Forthcoming.

[Koll and Kaindl, 1993] A.L. Koll and H. Kaindl. Bidirectional best-first search with bounded error: Summary of results. In *Proc. IJCAI-93,* pages 217 223, Chambery, France, 1993.

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

[Korf, 1993] R.E. Korf. Linear-space best-first search. *Artificial Intelligence,* 62(l):41-78, 1993.

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

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

[Pohl, 1971] I. Pohl. Bi-directional search. In *Machine Intelligence 6,* pages 127 140, Edinburgh, 1971. Edinburgh University Press.

[Rao *et ai,* 1991] V.N. Rao, V. Kumar, and R.E. Korf. Depth-first vs best-first search. In *Proc. AAAI-91* pages 434-440, Anaheim, 1991. Los Altos, CA.: Kaufmann.

[Reinefeld and Marsland, 1994] 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.

[Russell, 1992] S. Russell. Efficient memory-bounded search methods. In *Proc. ECAI-92,* pages 1-5, Vienna, Austria, 1992. Chichester: Wiley.

[Sen and Bagchi, 1989] A.K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proc. IJCAI-89,* pages 297 302,1989.

[Sen and Bagchi, 1993] A.K. Sen and A. Bagchi. Job sequencing with quadratic penalties: an A*-based graph search approach. In *Proc. CAIA-93,* pages 190 196, Orlando, FL, March 1993.

[Taylor and Korf, 1993] L.A. Taylor and R.E. Korf. Pruning duplicate nodes in depth-first search. In *Proc. AAAI-93,* pages 756-761, Washington, D.C., 1993. Los Altos, CA.: Kaufmann.

[Townsend, 1978] W. Townsend. The single machine problem with quadratic penalty function of completion times: a branch-and-bound solution. *Management Science,* 24(5):530-534, 1978.

[Zhang and Korf, 1993] W. Zhang and R.E. Korf. Depth-first vs. best-first search: new results. In *Proc. AAAI-93,* pages 769-775, Washington, D.C., 1993. Los Altos, CA.: Kaufmann.