

# Dynamic Improvements of Heuristic Evaluations during Search

Gerhard Kainz and Hermann Kaindl

Siemens AG Österreich  
 Geusaugasse 17  
 A-1030 Wien  
 Austria — Europe

## Abstract

Heuristic search algorithms employ evaluation functions that utilize heuristic knowledge of the given domain. We call such functions *static* evaluation functions when they only make use of knowledge applied to the given state but *not* results of any search in this state space — neither the search guided by the evaluation nor extra search like look-ahead. Static evaluation functions typically evaluate with some error. An approach to improve on the accuracy of a given static evaluation function is to utilize results of a search. Since this involves dynamic changes, we call resulting functions *dynamic* evaluation functions. We devised a new approach to dynamic improvements that we named *difference* approach. It utilizes differences of known costs and their heuristic estimates from a given evaluation function to improve other heuristic estimates from this function. This difference approach can be applied in a wide variety of known search algorithms. We show how it fits into a unifying view of dynamic improvements, that also covers already existing approaches as viewed from this perspective. Finally, we report experimental data for two different domains that represent significant improvements over previously published results.

## Notation

$s, t$	Start node and goal node, respectively.
$d$	Current search direction index; when search is in the forward direction $d = 1$ , and when in the backward direction $d = 2$ .
$k_i(m, n)$	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 $n$ to $t$ if $i = 2$ .
$h_i^*(n)$	Cost of an optimal path from $n$ to $t$ if $i = 1$ , or from $s$ to $n$ if $i = 2$ .
$g_i(n), h_i(n)$	Estimates of $g_i^*(n)$ and $h_i^*(n)$ , respectively.
$h^*(m, n)$	Cost of an optimal path from $m$ to $n$ .
$h(m, n)$	Estimate of $h^*(m, n)$ .
$f_i(n)$	Static evaluation function: $g_i(n) + h_i(n)$ .
$H_i(n)$	Dynamic estimate of $h_i^*(n)$ .
$F_i(n)$	Dynamic evaluation function: $g_i(n) + H_i(n)$ .

## Background and Introduction

Heuristic search algorithms employ evaluation functions that utilize heuristic knowledge of the given domain. Evaluation functions for problem solving (in contrast to two-player games) typically use heuristic estimates of the minimal cost of a path from the node evaluated to some target node. If the heuristic function never overestimates the real cost, it is said to be *admissible*, which is an important property for guaranteeing optimal solutions. Theoretically, a stricter property called *consistency*<sup>1</sup> is also very important, and in practice admissible functions are normally also consistent (Pearl 1984). We call such functions *static* evaluation functions when they only make use of knowledge applied to the given state but *not* results of any search in this state space — neither the search guided by the evaluation nor extra search like look-ahead. A usual example of such a static evaluation function to be found in text books is the Manhattan distance heuristic in the well-known 15-Puzzle, which is consistent and therefore also admissible.

Static evaluation functions typically evaluate with some error, i.e., the difference between the minimal cost of a path and its heuristic estimate is in most cases greater than zero. An approach to improve the accuracy of a given static evaluation function is to perform a search and to utilize its results. Since this involves dynamic changes, we call it a *dynamic* evaluation function.

We can view several search algorithms (like RTA\* (Korf 1990), MREC (Sen & Bagchi 1989), SMA\* (Russell 1992), ITS (Ghosh, Mahanti, & Nau 1994), or Trans (Reinefeld & Marsland 1994)) as approaches to using dynamic evaluation functions. They back up values during their normal search, which are cached in stored nodes. When such a node is re-searched, an improved backed-up value can often be used instead of the value assigned directly by the static evaluation function.

We devised a new approach to dynamic improvements that we call *difference* approach. It utilizes dif-

<sup>1</sup> $h_i(m) \leq h_i(n) + k_i(m, n)$  for all nodes  $m$  and  $n$ .

ferences of known costs and their heuristic estimates from a given evaluation function to improve other heuristic estimates from this function. This difference approach can be applied in a wide variety of known search algorithms. It is exemplified in two new methods for dynamic improvements of heuristic evaluations during search.

First, we present our new difference approach to dynamic improvements of heuristic evaluations during search and its use in various search algorithms. Then we show how it fits into a unifying view of dynamic improvements, that also covers already existing approaches as viewed from this perspective. Finally, we report experimental data for two different domains that represent significant improvements over previously published results.

## The New Difference Approach

Our new approach to dynamic improvements of heuristic evaluations during search is based on differences between known costs and heuristic estimates. Such differences are utilized by two concrete methods as presented below.

The basic idea common to these methods is that for many nodes during the search, the actual cost of a path to or from them is already known. Since the static heuristic values can normally be gained rather cheaply, the *differences* can be computed that signify the error made in their evaluation compared to the cost of the known path. These differences are utilized to improve other heuristic estimates during the same search.

These methods can be employed in a wide variety of known search algorithms. We present our new methods and their implementation in some important search algorithms.

### The Add method

The first method instantiates this approach by *adding* a constant derived from such differences to the heuristic values of the static evaluation function. Therefore, we call it the *Add* method.

Note, that adding a constant to all evaluations does not change the order of node expansions in a unidirectional search algorithm like A\* (Hart, Nilsson, & Raphael 1968). So, the benefit from this approach may not be immediately obvious. However, in most bidirectional search algorithms, estimates are compared to the cost of the best solution found so far (which is not necessarily already an optimal one), and having better estimates available for such comparisons improves the efficiency due to earlier termination. We explain how we apply this approach in various search algorithms in more detail below.

Let us use Fig. 1 to explain the key idea of this method. We assume consistency of the static heuristic evaluator  $h_i$ . Around the target node  $t$ , a search has examined a part of the graph and stored all the optimal paths from nodes  $B_i$  on its closed fringe to

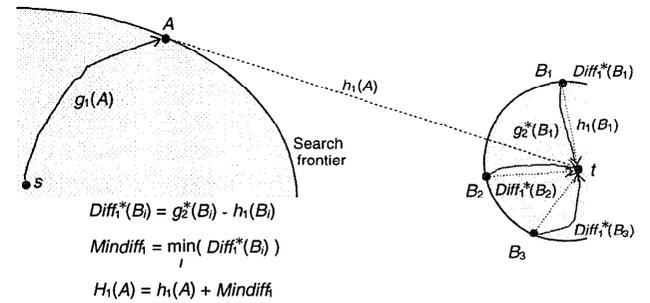


Figure 1: An illustration of the *Add* idea.

$t$ . For each node  $B_i$ , its heuristic value  $h_1(B_i)$  is computed and subtracted from the optimal path cost  $g_2(B_i) = g_2^*(B_i) = h_1^*(B_i)$ , resulting in  $Diff_1^*(B_i)$ . This is actually the error made by the heuristic evaluation of node  $B_i$ . The minimum of  $Diff_1^*(B_i)$  for all nodes  $B_i$  on the fringe is computed — we call it  $Mindiff_1$ .

The point of the *Add* method is that any consistent heuristic value  $h_1(A)$  for some node  $A$  outside this stored graph underestimates  $h_1^*(A)$  by at least  $Mindiff_1$ . We prove this precisely below, but first we need to show a result about  $Diff_1^*$ .

**Lemma 1** *If the heuristic  $h_1$  is consistent, then on any optimal path from some node  $n$  to  $t$  with an intermediary node  $m$*

$$Diff_1^*(m) \leq Diff_1^*(n)$$

*holds, i.e.,  $Diff_1^*$  never decreases on an optimal path with increasing distance from the target node  $t$ .*

**Proof:** If the heuristic  $h_1$  is consistent, then we have

$$h_1(n) \leq h_1(m) + k_1(n, m)$$

From this we simply obtain

$$g_2^*(m) - h_1(m) \leq g_2^*(m) + k_1(n, m) - h_1(n)$$

Since  $n$  and  $m$  are on one optimal path to  $t$ , we know that

$$g_2^*(n) = g_2^*(m) + k_1(n, m)$$

After substitutions we obtain

$$g_2^*(m) - h_1(m) \leq g_2^*(n) - h_1(n)$$

and equivalently

$$Diff_1^*(m) \leq Diff_1^*(n)$$

which proves the lemma.  $\square$

**Theorem 1** *If the heuristic  $h_1$  is consistent, then it is possible to compute an admissible heuristic  $H_1$  for some node  $A$  outside the search frontier around  $t$  by*

$$H_1(A) = h_1(A) + Mindiff_1 \leq h_1^*(A)$$

**Proof:** When some path exists from node  $A$  to  $t$ , also an optimal path must exist, and let it go through the frontier node  $B_j$ . (If no such path exists,  $h_1^*(A)$  is infinite and the theorem holds.) From Lemma 1 and the definition of  $Mindiff_1$  we know that

$$Mindiff_1 \leq Diff_1^*(B_j) \leq Diff_1^*(A)$$

Since  $Diff_1^*$  is the error made by the heuristic  $h_1$ , we can write

$$h_1(A) + Diff_1^*(A) = h_1^*(A)$$

After substitution we obtain

$$h_1(A) + Mindiff_1 \leq h_1^*(A)$$

which proves the theorem.  $\square$

**Corollary 1**  $H_1(A)$  is also an admissible estimate if  $A$  is a frontier node.

**Proof:** We can replace  $A$  by  $B_j$  in the proof of Theorem 1 without changing its validity.  $\square$

**Theorem 2** If the heuristic  $h_1$  is consistent, then  $H_1(A)$  is consistent.

**Proof:** If the heuristic  $h_1$  is consistent, then we have

$$h_1(n) \leq h_1(m) + k_1(n, m)$$

Adding the constant  $Mindiff_1$  on both sides leads to

$$h_1(n) + Mindiff_1 \leq h_1(m) + Mindiff_1 + k_1(n, m)$$

This means that

$$H_1(n) \leq H_1(m) + k_1(n, m)$$

which proves the theorem.  $\square$

Now let us sketch how this *Add* method can be utilized in some important known search algorithms. When using it based on a unidirectional algorithm like  $A^*$ , it is necessary to perform an additional search from  $t$  before the “normal” search starts from node  $s$ , in order to get some value  $Mindiff_1$ . If this search around  $t$  is also guided by a best-first search like  $A^*$ , optimal paths to all nodes within the search frontier are guaranteed but not to all frontier nodes themselves. If a suboptimal path was found to some frontier node, however, it is known that an optimal path leads through another frontier node with an optimal path to  $t$ . So, this does not change  $fmin$ , since the costs of suboptimal paths cannot influence the minimum.

Of course, a larger value of  $Mindiff_1$  is to be preferred for a given amount of search. So, this search is actually guided by expanding always one of those nodes  $n$  with minimal  $Diff_1^*(n)$ . Afterwards,  $Mindiff_1$  is during the normal search just a constant to be added.

It is also necessary to check, whether a node to be evaluated is outside or on the fringe of the graph around  $t$ . This is simply achieved through hashing, which is to be done anyway. When a node on the fringe is matched, a solution is already found, and when the first node of a path inside the stored graph around  $t$  is matched, this path need not be pursued any further, since its optimal continuation is already known. So, only the evaluator  $H_1$  is actually used, which is consistent, and therefore  $A^*$  does not have to re-open nodes (Pearl 1984). The search terminates when it selects some node  $n$  for expansion with  $f_1(n) = g_1(n) + H_1(n)$  being equal to the cost of the best solution found so far, which is proven this way to be an optimal one. We call the resulting algorithm *Add- $A^*$* .

Analogously to  $A^*$ ,  $IDA^*$  can also utilize the *Add* method. Again, it is necessary first to perform an ad-

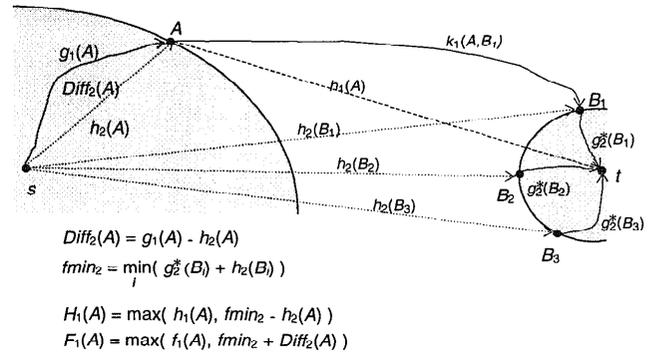


Figure 2: An illustration of the *Max* idea.

ditional search from  $t$ , that has an extra storage requirement for this graph. In this graph  $IDA^*$  must perform hashing in addition to its normal procedure. When using such storage around  $t$ , it is easily possible to utilize it also for another purpose — finding a solution earlier as implemented in *BAI* (Kaindl *et al.* 1995). We call the resulting algorithm *Add-BAI*.

For more details on this method, its theoretical properties and its use in various search algorithms we refer the interested reader to (Kainz 1994).

## The *Max* method

The second method computes its own estimate based on such differences and uses the *maximum* of this and the static estimate. Therefore, we call it the *Max* method.

Let us use Fig. 2 to explain the key idea of this method. We assume consistency of the static heuristic evaluator  $h_i$ , and that a path from  $s$  to  $A$  with cost  $g_1(A)$  is known. So we know for its evaluation of node  $A$ :  $h_2(A) \leq g_1(A)$ . The difference is  $Diff_2^*(A) = g_1(A) - h_2(A)$ . We use this difference for the construction of an admissible estimate  $F_1(A)$  of the cost of an optimal path from  $s$  to  $t$  that is constrained to go through  $A$ . Note, that  $g_1(A) = g_1^*(A)$  is *not* necessary, so we call the difference used here  $Diff_2^*(A)$  instead of  $Diff_2^*(A)$ .

In addition, we assume that a search has been performed from  $t$  in the reverse direction. From this search, we assume that from all frontier nodes  $B_i$  optimal paths to  $t$  are known, with cost  $g_2^*(B_i)$ . Therefore, it is possible to compute

$$fmin_2 = \min_i (g_2^*(B_i) + h_2(B_i))$$

Based on these assumptions, we can again construct a dynamic evaluation function as follows.

**Theorem 3** If the heuristic  $h_1$  is consistent, then it is possible to compute an admissible heuristic  $h'_1$  for some node  $A$  outside the search frontier around  $t$  by

$$h'_1(A) = fmin_2 - h_2(A) \leq h_1^*(A)$$

**Proof:** Every path from  $A$  to  $t$  must go through some frontier node  $B_j$ . The cost  $C_j$  of any such path is

bounded from below as follows:

$$C_j \geq k_1(A, B_j) + g_2^*(B_j)$$

If  $h_1$  is consistent, it is possible to estimate the optimal cost of a path between two nodes through

$$k_1(A, B_j) \geq h_2(B_j) - h_2(A)$$

Therefore, we can write

$$C_j \geq h_2(B_j) - h_2(A) + g_2^*(B_j)$$

Since  $fmin_2 = \min_i(g_2^*(B_i) + h_2(B_i))$ , we can also write

$$C_j \geq fmin_2 - h_2(A)$$

This is valid for the cost of any path from  $A$  to  $t$  including an optimal one, and so we can conclude

$$h_1^*(A) \geq fmin_2 - h_2(A)$$

which proves the theorem.  $\square$

**Corollary 2**  $h_1^*(A)$  is also an admissible estimate if  $A$  is a frontier node.

**Proof:** We can replace  $A$  by  $B_j$  in the proof of Theorem 3 without changing its validity.  $\square$

This dynamic evaluation function is not necessarily better for all nodes than the static function, and so it is useful to combine these functions:

$$H_1(A) = \max(h_1(A), fmin_2 - h_2(A))$$

Since both are admissible the resulting function is also admissible. When the value  $fmin_2$  changes during the search, however,  $H_1$  is not consistent.

In order to show how the difference  $Diff_2(A) = g_1(A) - h_2(A)$  is involved in this method, we also derive here the overall evaluation function

$$F_1(A) = \max(f_1(A), fmin_2 + Diff_2(A))$$

Now let us sketch how this *Max* method can be utilized in some important search algorithms. When using it based on a unidirectional algorithm, it is again necessary to perform an additional search from  $t$  in order to get some value  $fmin_2$ . This search can be performed either before or concurrently to the “normal” search starting from node  $s$ . Again, like in the *Add* method, it is not necessary that optimal paths from  $t$  to all frontier nodes are known. For getting values  $fmin_2$  that are as large as possible for a given amount of search, the usual strategy of selecting a node with minimal  $f_2$  is appropriate here. During the normal search, it is again necessary to check, whether a node to be evaluated is outside or on the fringe of the graph around  $t$ .

IDA\* can utilize the *Max* method similarly to the *Add* method in the compound search algorithm BAI. It must again perform hashing in the graph stored around  $t$ , as BAI does anyway. We call the resulting algorithm Max-BAI. When a transposition table (Reinefeld & Marsland 1994) is used in addition as in BAI-Trans (Kaindl *et al.* 1995), we call it Max-BAI-Trans.

Most interestingly, IDA\* can also utilize the *Max* method *without additional storage requirements*. Let us sketch the basic approach for such a *linear-space* application of this method here. While IDA\* tradition-

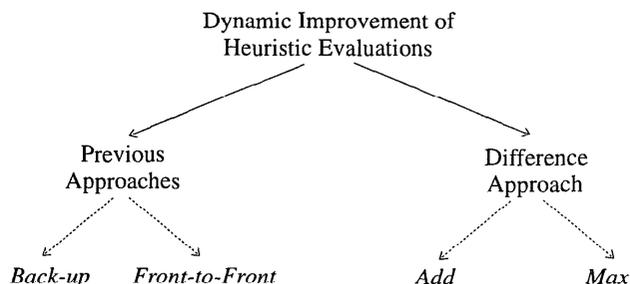


Figure 3: A taxonomy of methods for dynamic improvement.

ally searches in one direction only, we let it alternate the search direction. It computes  $fmin_i$  in one iteration to be used in the subsequent iteration, which must search in the alternate direction so that it can use this value. For example, in an iteration searching from  $t$  to  $s$ , the adapted IDA\* computes  $hmax_1 = \max(h_1(B_i))$  for all nodes  $B_i$ . This value is used as an estimate in the subsequent iteration for checking, whether a node  $A$  to be evaluated is “outside”: if  $h_1(A) > hmax_1$  is true, then node  $A$  cannot be “inside” and  $H_1(A)$  can be safely used. This check substitutes hashing in a stored graph. Since the static heuristic function normally underestimates, however, for some nodes  $H_1$  is not used although it would theoretically be correct to use it. We call the resulting algorithm that is based on this idea Max-IDA\*.

For more details on this method, its theoretical properties and its use in various search algorithms we refer the interested reader to (Kainz 1996).

## Dynamic Improvements of Heuristic Evaluations

As indicated above, these two methods according to our new difference approach are instances of a more general theme — dynamic improvements of heuristic evaluations during search. Fig. 3 shows a taxonomy of such methods, that includes previous approaches as well. We categorize them as the *Back-up* and the *Front-to-Front* methods, respectively. Let us illustrate them in our context and shortly discuss advantages and disadvantages of all the methods we are aware of — including our new ones.

The idea of the *Back-up* method is illustrated in Fig. 4. When during the normal search nodes  $B_i$  are statically evaluated but not stored, these values can still be used by backing them up to some node  $A$  that is stored. The *dynamic* value of  $A$  is the minimum of the estimated costs of the best paths found through the nodes  $B_i$ . Unless the static evaluator is consistent, it is useful to store the maximum of the dynamic and the static value of a node. When such a cached node is researched, an improved value can often be used instead of the value assigned directly by the static evaluation function.

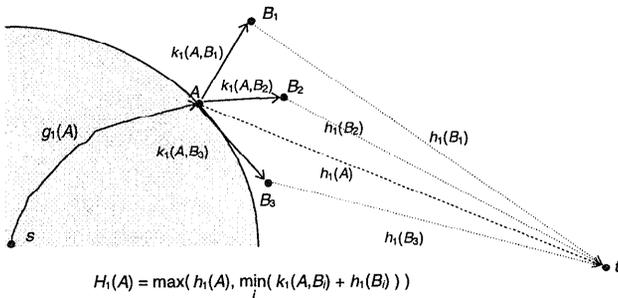


Figure 4: An illustration of the *Back-up* idea.

This Back-up method is widely applied in many algorithms like RTA\* (Korf 1990), MREC (Sen & Bagchi 1989), MA\* (Chakrabarti *et al.* 1989), SMA\* (Russell 1992), ITS (Ghosh, Mahanti, & Nau 1994) and Trans (Reinefeld & Marsland 1994). Its advantages are very little overhead and steady (though often modest) improvement with increasing memory size. In addition, it also works when a goal condition instead of a goal node is specified, i.e., it does not require that a goal node is explicitly given. However, it is only applicable for re-searched and cached nodes, and we cannot see how it could make sense in the context of traditional best-first search like A\*.

The idea of the *Front-to-Front* method is illustrated in Fig. 5. When for the evaluation of some node A nodes  $B_i$  on the opposite search front are available in storage, the costs of optimal paths from A to every  $B_i$  can be estimated. Adding these to known costs of paths from  $B_i$  to the target  $t$ , normally more accurate dynamic estimates can be gained than from the static evaluator.

This Front-to-Front method is applied in BHFFA (de Champeaux & Sint 1977; de Champeaux 1983), and more recently in *perimeter search* (Dillenburg & Nelson 1994) with a run-time optimization presented in (Manzini 1995). These algorithms perform a *wave-shaping* strategy. Since for the selection of a node in one search frontier typically all its nodes must be evaluated, heuristic estimates between all nodes in this one frontier and all nodes in the other must be computed. That is, the effort is proportional to the cross product of the numbers of nodes in the frontiers. So, the advantage of improved evaluation accuracy is to be balanced with this large overhead in time consumption. While BHFFA can for this reason only find optimal solutions to quite easy problems, perimeter search is comparably cheaper, since one of its fronts is constant. It typically has an optimum in time efficiency for low perimeter depths. The run-time optimization presented in (Manzini 1995) shifts the optimum to higher perimeter depths which save more node generations. This makes it successful for the 15-Puzzle (only 15-Puzzle results are published yet). However, as we will show below, for other domains no such success is guaranteed,

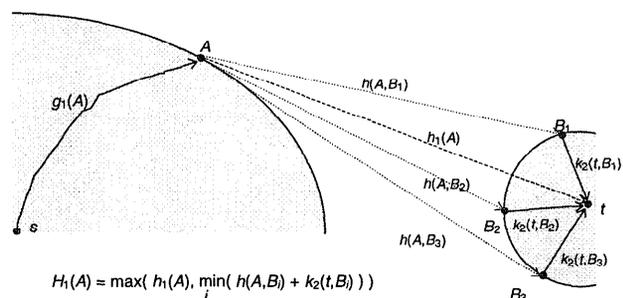


Figure 5: An illustration of the *Front-to-Front* idea.

and the applicability of this optimization in traditional best-first search like A\* is hindered by large memory requirements for each stored node.

Now let us informally discuss our new *difference* approach in this context. The overhead of the *Add* method per node searched is negligible, and also that of the *Max* method is small. In principle, both methods are widely applicable (though they will not be of much use in a wave-shaping algorithm). The main disadvantage of the *Add* method appears to be that in case there are very few distinct values (as in the 15-Puzzle when using the Manhattan distance heuristic) even achieving a value  $Mindiff_1 > 0$  may require very much memory. In most cases, however, our new difference approach effectively utilizes known errors in heuristic evaluation to improve new heuristic evaluations.

In some sense, it is also possible to view our difference approach as *learning*, since also there differences between predicted and actual outcomes are important. Usual machine research, however, strives for using the results from one problem instance for solving subsequent instances, which we did not attempt. An in-depth discussion of this relationship is outside the scope of this paper.

## Experimental Results

### 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.<sup>2</sup> We compare algorithms that achieve the previously best results in this domain with one of these algorithms (BAI-Trans) enhanced by our new *difference* approach to dynamic improvements of heuristic evaluations. All the compared algorithms use no domain-specific knowledge about the puzzle other than the Manhattan distance heuristic. The main storage available on a Convex C3220 was up to 256 Mbytes.

Fig. 6 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

<sup>2</sup>We used the complete set of 100 instances from (Korf 1985).

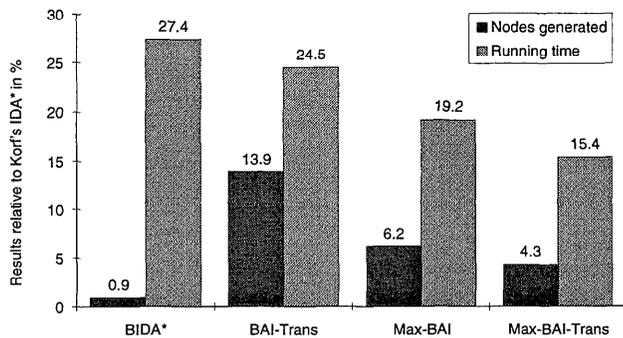


Figure 6: Comparison on the 15-Puzzle (100 instances).

respective search effort of IDA\* (in Korf's implementation), since it was the first algorithm able to solve random instances of the 15-Puzzle.

BIDA\* (Manzini 1995) — using the *Front-to-Front* method — generates just 0.9 percent of the number of nodes generated by IDA\*. However, it needs 27.4 percent of IDA\*'s running time.<sup>3</sup> In the given 256 Mbytes of storage, BIDA\* can store a maximum of 1 million perimeter nodes. This would correspond to a perimeter depth of 19, where BIDA\* generates just 0.4 percent of the number of nodes generated by IDA\*, but needs 42 percent of IDA\*'s running time. So, the running time has a minimum at some perimeter depth, and we show it in Fig. 6. It corresponds to a perimeter depth of 16, but we do not claim that this is the best perimeter depth in all implementations.

BAI-Trans generates more nodes (13.9 percent of IDA\*), but since its overhead per node is much smaller than that of BIDA\*, its running time is even slightly better (24.5 percent).<sup>4</sup>

Max-BAI and Max-BAI-Trans — both utilizing our new difference approach — achieve the fastest searches. They can store a maximum of 5 million nodes in our implementation of these algorithms in the given 256 Mbytes of storage. The fastest one is Max-BAI-Trans, needing just 15.4 percent of the time needed by IDA\*.

<sup>3</sup>BIDA\*'s result here is worse than the data reported by Manzini (Manzini 1995). This is primarily due to a different implementation that is based on the very efficient code of IDA\* for the puzzle by Korf that we are using. In such an implementation the overhead especially of wave shaping shows up more clearly even when using the run-time optimizations described in (Manzini 1995). While we had no access to the implementation by Manzini, in email communication with him we were given some hints about it, and there was mutual agreement about the overall effect on the relative running times due to the different implementations of IDA\*.

<sup>4</sup>The better results of BAI-Trans here as compared to (Kaendl *et al.* 1995) are due to an increase of the given storage by a factor of 20, where most of this storage is given to the Trans part.

For achieving this result, it uses 4 million nodes for the *Max* method (and BAI) and 1 million nodes for Trans. In comparison, Max-BAI as shown here uses just the 4 million nodes for the *Max* method (and BAI), so we can show the influence of Trans, which is comparably modest.

Our best variant of Max-IDA\* generates 54.4 percent of the number of nodes generated by IDA\*, and it needs 76.1 percent of IDA\*'s running time. Obviously, it cannot really compete with the algorithms that use additional space, but this result is good for a linear-space algorithm.

In summary, our *Max* method improves known search algorithms, and it led 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. The superiority of Max-BAI and Max-BAI-Trans in terms of running time over their competitors that do not use this approach is statistically significant. For more details on these results see (Kainz 1996).

## Maze

In order to get a better understanding of the usefulness of our new approach, we made also experiments in a second domain — finding shortest paths in a maze.<sup>5</sup>

We compare known algorithms that achieve the best results in this domain (as far as we found) with one of these algorithms (A\*) enhanced by our new *difference* approach to dynamic improvement of heuristic evaluations. The comparison is in terms of the average number of node generations and the running times. The data are normalized to the respective search effort of A\*.

PS\* (Dillenburg & Nelson 1994) — using perimeter search (*the Front-to-Front method*) — generates 99.3 percent of the number of nodes of A\*, but it needs 119.8 percent of the time used by A\*. These data correspond to a perimeter depth of 25, and they are the best we achieved with this method in terms of running time. We investigated larger perimeter depths (50, 100, ..., 250), and for these the numbers of generated nodes only marginally improve (up to 93.9 percent), while the running time becomes quite high (up to

<sup>5</sup>Our use of this domain was inspired by its use in (Rao, Kumar, & Korf 1991). Problem instances in this domain model the task of navigation in the presence of obstacles. They were drawn randomly using the approach behind the Xwindows demo package Xmaze. As a heuristic evaluator, we use the Manhattan distance as in (Rao, Kumar, & Korf 1991).

For our experiments, we made the following adaptations. In order to allow transpositions (they arise when different paths lead to the same node), we do not “install” a wall in three percent of the cases. This leads to roughly the same “density” of transpositions as in the 15-Puzzle. Moreover, we use much larger mazes — 2000 × 2000, and in order to focus on the more difficult instances of these, we only use instances with  $h_1(s) \geq 2000$ .

358.7 percent). We think that the running time could be reduced for perimeter depths smaller than 25, but for these no real savings in the number of nodes generated and therefore no improvement over A\* can be expected.

Apparently, the perimeter idea (as represented here by PS\*, since BIDA\* based on IDA\* is very inefficient on these mazes due to the high number of iterations) does not really work in this domain. In contrast, for the 15-Puzzle just some few perimeter nodes improve the static evaluation, since the (unit) costs of their arcs are often simply added. This has a large effect in this domain where the heuristic values are typically smaller than about 40. In the maze instances of the size we experimented with, the heuristic values are two orders of magnitude larger, and therefore many more perimeter nodes would be required to achieve much effect. These, however, make perimeter search very expensive both in terms of running time and probably also in storage requirement. A more in-depth investigation of the advantages and disadvantages of perimeter search can be found in (Kainz 1996).

We achieved the fastest searches in our maze experiments with Add-A\*. It generates 70.7 percent of the number of nodes of pure A\* and uses 71.7 percent of the time. These data are statistically significant improvements over A\* in both terms. Since also the application of our *Max* method achieves significant improvements in this domain (as reported in (Kainz 1996)), we conclude that our new *difference* approach in general is useful in this domain.

## Conclusion

*Static* heuristic evaluation functions typically evaluate with some error, but *dynamic* evaluation functions utilizing results of a search can reduce this error. We devised a new approach to dynamic evaluation that we named *difference* approach. It utilizes differences of known costs and their heuristic estimates from a given evaluation function to improve other heuristic estimates from this function. This approach is exemplified in two new methods, and we show how it fits into a unifying view of dynamic improvements, that also covers already existing approaches as viewed from this perspective.

Our experimental data for two different domains represent significant improvements over previously published results. So, our new approach to dynamic improvements of heuristic evaluations during search appears to work well.

## Acknowledgments

Our implementations are based on the very efficient code of IDA\* and A\* for the Puzzle provided to us by Richard Korf. For some of our experiments we had a Convex C3220 at the computing center of the TU Vienna available. Finally, we acknowledge the useful comments by Wilhelm Barth, Roland Steiner and the AAAI reviewers.

## References

- Chakrabarti, P.; Ghose, S.; Acharya, A.; and DeSarkar, S. 1989. Heuristic search in restricted memory. *Artificial Intelligence* 41(2):197-221.
- 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.
- Dillenburg, J., and Nelson, P. 1994. Perimeter search. *Artificial Intelligence* 65:165-178.
- Ghosh, S.; Mahanti, A.; and Nau, D. 1994. ITS: an efficient limited-memory heuristic tree search algorithm. In *Proc. Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1353-1358.
- 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.
- Kaindl, H.; Kainz, G.; Leeb, A.; and Smetana, H. 1995. How to use limited memory in heuristic search. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 236-242.
- Kainz, G. 1994. Heuristische Suche in Graphen mit der Differenz-Methode. Diplomarbeit, Technische Universität Wien.
- Kainz, G. 1996. Heuristische Suche mit begrenztem Speicherbedarf. Doctoral dissertation, Technische Universität Wien. Forthcoming.
- Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97-109.
- Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189-212.
- Manzini, G. 1995. BIDA\*: an improved perimeter search algorithm. *Artificial Intelligence* 75:347-360.
- Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley.
- Rao, V.; Kumar, V.; and Korf, R. 1991. Depth-first vs best-first search. In *Proc. Ninth National Conference on Artificial Intelligence (AAAI-91)*, 434-440. Anaheim: Los Altos, CA.: Kaufmann.
- Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 16(12):701-709.
- 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.