Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)

Using Lookaheads with Optimal Best-First Search

Roni Stern Tamar Kulberis Ariel Felner
Information Systems Engineering
Deutsche Telekom Laboratories
Ben Gurion University
Beer-Sheva, Israel 85104
roni.stern@gmail.com,{kulberis,felner}@bgu.ac.il

Robert Holte
Computing Science Department
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
holte@cs.ualberta.ca

Abstract

We present an algorithm that exploits the complimentary benefits of best-first search (BFS) and depth-first search (DFS) by performing limited DFS lookaheads from the frontier of BFS. We show that this continuum requires significantly less memory than BFS. In addition, a time speedup is also achieved when choosing the lookahead depth correctly. We demonstrate this idea for breadth-first search and for A*. Additionally, we show that when using inconsistent heuristics, Bidirectional Pathmax (BPMX), can be implemented very easily on top of the lookahead phase. Experimental results on several domains demonstrate the benefits of all our ideas.

Introduction

Best-first search (BFS) is a general purpose search algorithm. It keeps a *closed list* (denoted hereafter as CLOSED) of nodes that have been expanded, and an *open list* (denoted hereafter as OPEN) of nodes that have been generated but not yet expanded. At each cycle of the algorithm, it expands the most promising node (the *best* node) from OPEN which is moved from OPEN to CLOSED, and its children are generated and added to OPEN. BFS terminates when a goal node is chosen for expansion, or when OPEN is empty.

Special cases of BFS differ in their cost function f. *Breadth-first search* (denoted here as BRFS) is a BFS with $f = d$ where d is the depth in the search tree. A* is a BFS with $f(n) = g(n)+h(n)$, where $g(n)$ is the sum of the edge costs from the start to node n and $h(n)$ is a heuristic estimation of the cost from node n to a goal. A* with an admissible heuristic returns optimal solutions.

The main drawback of BFS is its memory requirements. In order to choose the best node, BFS stores in memory all the *open* nodes. Additionally, BFS also stores in memory all the *closed* nodes in order to recognize a state that has already been generated and to enable solution reconstruction once a goal is reached. The space complexity of BFS therefore grows exponentially with the depth of the search. Consequently, BFS cannot solve difficult problems since for large state spaces it usually exhausts all the available memory before reaching a goal. In addition, the constant time per node in a typical expansion cycle is also rather heavy due to the different primitive operation that are performed.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

By contrast, the memory needs of *depth-first search* (DFS) is only linear in the depth of the search and the constant time per node is rather light. DFS in its basic form may never find a solution and if it does, it does not guarantee anything about the quality of the solution. However, several algorithms that generate nodes in a best-first fashion while activating different versions of DFS have been developed. The most common algorithm is IDA* (Korf 1985) which performs a series of DFS calls from the root in an increasing order of costs. Another related algorithm is RBFS (Korf 1993). It expands new nodes in a best-first order even when the cost function is *nonmonotonic*. The main limitation of DFS algorithms is that their efficiency significantly deteriorates in state spaces that contain many transpositions since they cannot perform duplicate pruning. In addition, they are inefficient with a many unique heuristic values.

We propose a continuum between classic BFS and DFS. The basic idea is to use the general schema of BFS but to perform limited DFS lookaheads from nodes when they are expanded. This schema has great potential to exploit the complimentary benefits of BFS and DFS. Using these lookaheads, significant reduction in the memory requirements is always obtained. Furthermore, when correctly choosing the lookahead depth we keep the duplicate nodes rate small enough and significant time speedup can be achieved too. We also describe how lookaheads can be effectively augmented with BPMX (Felner et al. 2005) when using inconsistent heuristics. Experimental results demonstrate the benefits of the presented algorithms on several domains.

Other attempts to search in a best-first order with a limited amount of memory include MA* (Chakrabarti et al. 1989) and SMA* (Russell 1992). These algorithms generate fewer nodes than IDA* but they run slower in practice because of memory maintenance overhead (Korf 1993). The most similar to our approach is MREC (Sen and Bagchi 1989). A* is activated, until memory is exhausted. Then IDA* is performed from the frontier. There is a significant difference between MREC and our approach as we activate the DFS phase at every level regardless of the memory available.

Breadth-first Search with Lookahead

BRFS with lookahead ($BRFSL(k)$) combines BRFS with limited DFS search to depth k. As in ordinary BRFS, BRFSL maintains both OPEN and CLOSED. At each itera-

Algorithm 1: Expansion cycle of BRFSL(k)

```
   Input: n, the best node in the open list
1  foreach state operator op do
2  |  child ← generateNode(op, n)
3  |  if DFS-with-GoalTest(child,k)=True then
4  |  |  halt ; /* Goal was found          */
5  |  if duplicateDetection(child)=False then
6  |  |  insert child to OPEN
7  |  else
8  |  |  update cost function of duplicate (if required)
9  |  end
10 |  insert n to CLOSED
11 end
```

tion, node n with smallest depth in OPEN is extracted. However, before adding n to CLOSED, a lookahead to depth k is performed by applying a limited DFS to depth k from n. Only nodes at depth k are goal tested.[1] If a goal is found during the lookahead, the algorithm halts immediately. If a goal is not found, node n is added to CLOSED, and its children are added to OPEN. Algorithm 1 presents the pseudo code for BRFSL. Line 3 performs a limited DFS from $child$. If a goal is found during this limited DFS, then *DFS-with-GoalTest* return $True$.

An exception to this is the first step of expanding the root. In this case we do not stop when a goal is reached but continue the DFS to verify that no goal exists at shallower levels. In fact, at this step, iterative deepening can be performed in order to either find a goal at depth *less than or equal to* k or to verify that no goal exists at these levels.

Three expansion cycles of BRFSL(2) are illustrated in Figure 1. Dark nodes indicate the expanded nodes, light nodes are the nodes visited during lookahead steps, and nodes with bold outlines were also goal tested. In the first expansion cycle (a), a DFS is performed from the root node A to depth 2. Since this is the first step, a goal test is performed for all nodes. Assuming no goal was found in this cycle, the algorithm will add the direct successors of the root, B and C, to OPEN. In the next iteration (b), node B is expanded. Instead of only generating its immediate successors (D and E), a lookahead to depth 2 is performed, where only nodes at the deepest lookahead level are goal tested (nodes H,I,J and K). Assuming no goal is found D and E are added to OPEN and a new iteration starts (c).

Completeness and Optimality

It is easy to see that $BRFSL(K)$ is complete, i.e., it always finds a goal if one exists. The algorithm also provides the optimal solution. Assume the goal node is at depth d and the lookahead is performed to depth k. When expanding nodes at depth $d-k$ we are actually peeking at nodes at depth d for the first time. Since nodes at depth smaller than d were peeked at in earlier expansions, when a goal node is found at depth d it is optimal.

[1] Throughout this paper we use the term *goal test* as a verb, denoting the process of checking if a given node is a goal node.

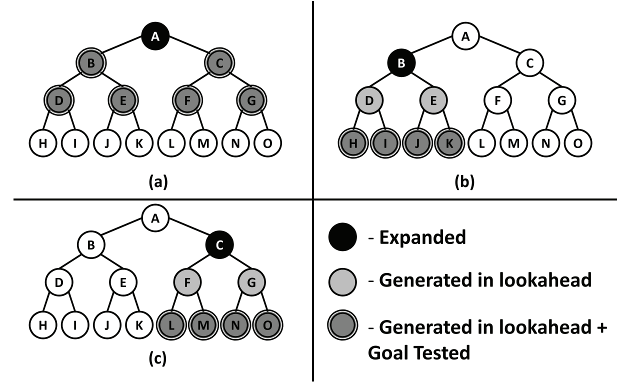

Figure 1: Example of BRFSL(2)

Memory complexity Assume that the *brute-force* branching factor of the state space is b and that b_e is the *effective* branching factor, i.e., the number of successors after applying duplicate pruning. It is important to note that $b_e \leq b$, due to transpositions in the state space. Now, assume that the depth of the solution is d and the lookahead is performed to depth k. When the goal node is found, only nodes up to depth $d-k$ are stored in OPEN and CLOSED. Thus, the worst case space complexity of $BRFSL(K)$ is

$$1 + b_e + b_e^2 + b_e^3 + \ldots + b_e^{d-k} = O(b_e^{d-k})$$

Clearly this requires less space than standard BRFS (= BRFSL(0)) which requires $O(b_e^d)$ memory.

Time complexity We differentiate between two types of nodes: *expanded* nodes and *visited* nodes. *Expanded* nodes are nodes that are expanded from OPEN. Based on the space complexity analysis above there are $1 + b_e + b_e^2 + b_e^3 + \ldots + b_e^{d-k}$ such nodes. *Visited* nodes are nodes that are visited during the lookahead DFS calls. A single DFS lookahead search to depth k visits $1 + b + b^2 + b^3 + \ldots + b^k = O(b^k)$ nodes. We perform such lookaheads for all expanded nodes and thus the total number of nodes that are visited during all DFS lookaheads is $(1 + b_e + b_e^2 + b_e^3 + \ldots + b_e^{d-k}) \times (1 + b + b^2 + b^3 + \ldots + b^k)$ this amounts to $O(b_e^{d-k} \times b^k)$. This is larger than the number of nodes traversed by ordinary BRFS ($= 1 + b_e + b_e^2 + b_e^3 + \ldots + b_e^d = O(b_e^d)$) for two reasons. First, unlike regular BRFS where every node is visited only once, in BRFSL(k) every node from depth larger than k and smaller than d is visited at least k times during the lookaheads of previous levels. Second, in the DFS lookaheads we do not prune duplicate nodes and if the state space contains transpositions, duplicate nodes are visited via different subtrees. In other words, for the DFS steps we use b as the base of the exponent while for the BRFS steps we always use b_e.

However, in practice $BRFSL(k)$ may run faster than BRFS. Since the lookahead is done in a depth-first manner, nodes can be generated during the lookahead by only applying the state space operator on the expanded node and applying the reverse operator when backtracking. This avoids the need to copy an entire state for every node in the lookahead, yielding an improved runtime. Therefore, the constant time per node during the lookahead may be much smaller than the constant time required to completely generate a node and insert it to OPEN. In addition, no duplicate detection (DD)

15-puzzle				
k	Memory	Expanded	DFS	Time
0	**19,473,242**	10373537	0	16.62
1	**11,305,305**	5978496	12566619	9.56
2	**5,978,495**	3138531	20674663	6.10
3	**3,138,530**	1634463	26483929	4.80
4	**1,634,462**	845867	31085133	**4.26**
5	**845,866**	434590	34966998	**4.25**
6	**434,589**	222132	38537864	**4.31**
7	**222,131**	112758	41900800	4.60
8	**112,757**	56961	45243397	4.78
9	**56,960**	28568	48437217	5.21
10	**28,567**	14258	51578154	5.37
11	**14,257**	7050	54304459	5.78
12	**7,049**	3467	56902134	5.92
13	**3,466**	1682	58866868	6.25
14	**1,681**	808	60560268	6.29
15	**807**	385	61653712	6.58
16	**384**	182	62489184	6.48
17	**181**	86	62764526	6.69
18	**85**	40	62911950	6.58
(12,4) TopSpin puzzle				
0	**12,575,891**	1845383	0	17.35
1	**1,854,382**	258032	3096386	3.31
2	**258,031**	34887	5442500	**2.99**
3	**34,886**	4592	8651828	4.5
4	**5,591**	583	13207502	6.68

Table 1: Results of BRFSL.

check is done at the DFS stage which also saves time since a DD check might be time consuming. Thus, for small values of k, $BRFSL(k)$ might run faster than BRFS.

When increasing k, more nodes are visited by $BRFSL(k)$ because of the two reasons above (duplicates and overlapping lookaheads). At some point, this will dominate the fact that the constant time per node is smaller. The optimal value for k is domain dependent and is strongly related to the rate of duplicates in the domain and to the constants involved.

Experimental Results

We experimented with BRFSL(k) on the 15-puzzle, for different values of k. Table 1 presents the results averaged over 50 depth-22 instances. The k column shows the lookahead depth (where $k = 0$ is ordinary BRFS). The *Memory* column shows the amount of memory required for search. The *Expanded* column shows the number of nodes expanded during the search. The DFS column shows the number of nodes visited during the DFS lookahead phase and the *time* column shows the runtime in seconds. As expected, larger values of k constantly reduce the memory needs (labeled in **bold**) compared to ordinary BRFS (BRFSL(0)). Furthermore, for all values of k the search was also faster than BRFSL(0). Optimal time behavior was obtained for $k = 5$ where the time speedup was a factor of 4 and the memory reduction was by a factor of 20.

Similar results were achieved for 50 instances of the TopSpin(12,4) puzzle at depth 14 (see results in the bottom of the table). In TopSpin, the optimal lookahead depth was 2 (compared to 5 in the 15-puzzle). This is because TopSpin contains many short cycles even in our implementation, which eliminates the most obvious transpositions.

A* with Lookahead

Generalizing the lookahead concept to work with A* is not trivial. In BRFS and in BRFSL(k) as well, all nodes of depth d are visited for the first time before visiting nodes of depth $d + 1$. When a goal node is found (during the lookahead phase) the search can halt, as it is guaranteed that the optimal path to it has been found. By contrast, when A* expands a node it generates new nodes with a variety of costs. Therefore, the best path to the goal is guaranteed only when all nodes with cost lower than a goal node have been expanded. Thus halting the search when a goal is seen in the lookahead may result in a non-optimal solution. We now show how a lookahead can be effectively applied in A*, while guaranteeing the optimal solution. The resulting algorithm is called A* with lookahead (AL*).

AL* uses the same basic expansion cycle of A*. The least cost node v from OPEN is expanded and moved to CLOSED, and its children are generated and added to OPEN. Similar to BRFSL, after a node v is expanded, a limited DFS lookahead is first performed from v. However, AL* has a few modifications:

Bounding the lookahead with the cost function: All DFS lookaheads performed in the expansion cycle of node v expand only nodes with costs *less than or equal to* $f(v) + k$, The DFS backtracks when a node with cost larger than $f(v) + k$ or a goal node is visited. We denote such nodes as the *lookahead frontier nodes*.

Maintaining an upperbound on the goal cost. In order to guarantee the optimality of the solution we use the same mechanism used by Depth-First Branch and Bound (DFBnB) (Korf 1993). We maintain a variable UB with the cost of the best solution found so far (initially ∞). The search halts when no node in OPEN has a cost *smaller* than UB. In addition, nodes with cost *larger than or equal to* UB can be immediately pruned as they cannot lead to a better solution.

Propagating cost found during the lookahead. We associate two f-values with each node. (1) f-static (f_s) which is its original f-value and (2) f-updated (f_u): once a lookahead is performed from a child c, we keep the smallest f-value among the *lookahead frontier nodes*. This value is a lower bound on any solution in the subtree rooted by c. f_u is used to order OPEN.[2] However, f_s is used to bound the cost of the lookahead. That is, the lookahead is performed up to cost $f_s + k$. Otherwise, using the f_u to bound the lookahead will result in a growing lookahead depths, that will increase the runtime as the search progresses.

Algorithm 2 describes an expansion cycle of AL^*. The search terminates when the cost of the expanded node v is *larger than or equal to* UB (lines 1-2). Child nodes with cost *larger than or equal to* UB are pruned (line 6-7). UB is updated either when a child is found to be a goal (line 9), or during the lookahead (line 12-13). The lookahead bound LHB is set to be the minimum between $f_s(v) + k$ and UB

[2]A reminiscent idea was also used in RBFS (Korf 1993; Reinefeld and Marsland 1994) where two f-values were used: *static* and *stored*.

Algorithm 2: Expansion cycle of A* with lookahead

```
   Input: v, the best node in the open list
   Input: UB, an upper bound, initialized with ∞
 1 if cost(v) ≥ UB then
 2 | Halt ; /* Optimality verified */
 3 Insert v to closed list
 4 foreach operator op do
 5 | child ← generateNode(op, v)
 6 | if f_u(child) ≥ UB then
 7 | | Continue ; /* Prune the node */
 8 | if goalTest(child)=True then
 9 | | UB=f_u(child)
10 | LHB ← min(UB, f_s(v) + k[, NextBest])
11 | if f_u(child) ≤ LHB then
12 | | MinCost ← Lookahead(child,LHB,UB,∞)
13 | | /* lookahead call possibly updating UB */
14 | | if MinCost > f(child) then
   | | f_u(child) ← MinCost
15 | if duplicateDetection(child)=False then
16 | | Insert child to open list
17 | else
18 | | Reorder child in OPEN (if required)
19 | end
20 end
```

Algorithm 3: AL^* Lookahead

```
   Input: v, the root of the lookahead
   Input: LHB, the limit of the lookahead
   Input: UB, an upper bound to the goal cost
   Input: MinCost, The min. cost of lookahead frontier node.
 1 foreach operator op do
 2 | child ← generateNode(op, v)
 3 | if goalTest(child)=True then
 4 | | UB=cost(child)
 5 | | MinCost ← min(MinCost, f_u(child))
 6 | else
 7 | | if f_u(child) ≥ LHB then
 8 | | | MinCost ← min(MinCost, f_u(child))
 9 | | else
10 | | | MinCost ← min(MinCost,
   | | | Lookahead(child,LHB, UB, MinCost))
11 | | end
12 | end
13 end
14 return MinCost
```

(line 10) (the square brackets is an optional version that will be discussed below). The lookahead DFS procedure (described in Algorithm 3 and called in line 12) may update UB, as well as the cost of $child$ (lines 14-15). $MinCost$ contains the minimum cost of a lookahead frontier node found so far. It is updated whenever a frontier node with smaller cost is found (lines 5,8 and 10 of Algorithm 3). Next we discuss several variants of the lookahead procedure.

Maintaining Best-First Order in the Lookahead

As explained above, if a node v with cost c is expanded then the lookahead step in $AL^*(k)$ performs a limited DFS up to cost $c + k$. Nodes inside this lookahead are visited in a DFS manner and not in a best-first manner. This might cause expansions of nodes with high costs even though the goal might be found later with a lower cost. Two possible reasons can cause this and both can be rectified.

First, if the next node v' in OPEN has a cost of $c_{next} < c + k$ then the DFS lookahead may expand nodes with costs larger than c_{next}. Thus $AL^*(k)$ does not necessarily visit new nodes in a best-first manner. This can be rectified by limiting the depth of the lookahead to be no larger than the cost of the next best node in OPEN. This option is shown in the square brackets in line 10 of Algorithm 2. This is reminiscent of the use of the next best alternative cost of RBFS (Korf 1993) (denoted there as the *upper bound*).

Second, the lookahead itself is performed in a DFS manner. Consequently, new nodes within the lookahead subtree with high costs (e.g., $c + k$) might be expanded before new nodes with smaller costs (e.g., $c + k - 1$). To rectify this, IDA*or RBFS can be used to implement a completely best-first lookahead. However, our algorithm produces the optimal solution even without these modifications. In the experimental results presented in this paper we have not implemented this due to the overhead of IDA* and RBFS.

Lookahead 0: Implementing Trivial Lookahead

In BRFS the cost function is the depth of the node. Thus, when expanding a node v with cost c we always generate nodes with cost $c + 1$. Therefore BRFSL with lookahead zero is simply BRFS. By contrast, in A* the cost function is ($f = g + h$). Therefore if for a child node x, h decreased by the same amount that g increased, the total cost of x will be equal to the cost of v. Thus, in AL*(0) a lookahead will be preformed to all nodes below v with the same cost of v. After the lookahead is completed a child node enters OPEN, but can then be immediately expanded. Inserting to OPEN is a costly operation as priority queues times are logarithmic in the size of OPEN. Thus, we suggest a simple enhancement to AL^*. If a child node has the same cost as its parent (which has just been expanded) immediately expand it, without inserting it to OPEN. We present this improvement in the context of $AL^*(0)$ but it can also be implemented in classic A* and in $AL^*(k)$ for $k > 0$. In domains where many successors share the same cost value, this enhancement is expected to substantially reduce execution time, as many nodes will bypass OPEN compared to classic A*. Note that this enhancement makes BFS more like BRFS, where each expanded node generates nodes with the next largest cost. We denote this enhancement as *trivial* lookahead and regard it as part of lookahead zero. This enhancement was recently introduced independently (Sun et al. 2009).

Figure 2 illustrates an expansion cycle of $AL^*(4)$ with trivial lookahead, when node A is chosen for expansion. First a trivial lookahead is performed, expanding all successors of A that have the same cost as A (=22). These are nodes C, F, and J, marked with black circles. Each child of these nodes with a cost larger than 22 is inserted to OPEN, and a lookahead starts from that node. These nodes are B, K, G, and D, marked with dark gray circles. During a lookahead, all successors with cost ≤ 26 (22+4) are visited

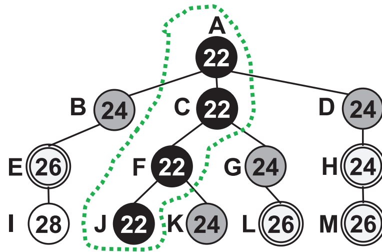

Figure 2: Example of an iteration of $AL^*(4)$.

Algorithm	Memory	Expanded	DFS	Time
15-puzzle (average solution length: 50.95)				
A*	4,790,944	2,653,442	0	15.23
AL*(0)	4,790,944	2,653,442	0	8.07
AL*(2)	1,371,783	994,645	8,752,976	2.72
AL*(4)	361,399	245,900	15,297,728	**1.60**
AL*(6)	91,482	59,083	22,418,926	1.68
AL*(8)	20,920	13,006	28,149,641	1.97
IDA*	-	-	34,314,189	4.22
(16,4) Top Spin (average solution length: 13.91)				
A*	6,582,232	889,802	0	63.58
AL*(0)	2,678,211	278,279	0	42.68
AL*(1)	829,494	101,896	14,299,225	**31.46**
AL*(2)	160,873	18,668	26,466,951	37.39
AL*(3)	86,962	14,131	35,344,526	48.54
IDA*	-	-	63,682,787	84.01

Table 2: AL^* results

and goal tested. Thus the nodes visited during the lookahead are E, L, H, and M, marked with a double circle.

Experimental Results

We implemented AL* on the 15-puzzle with Manhattan distance heuristic and on the (16,4)-TopSpin with a PDB heuristic based on 8 tokens. Table 2 presents the results. For the 15-puzzle we used the standard 100 random instances (Korf 1985). We report average results only for 74 instances that were solvable with A* using 2Gb of RAM. Due to less memory requirements of AL*(k), 90 instances were solvable with AL*(2), 98 with AL*(4) and the entire set with deeper lookaheads. For TopSpin the results are for 100 random instances. AL*(0) denotes the trivial lookahead. The meaning of the columns are the same as in Table 1.

In both domains using AL^* with larger lookaheads yields substantial improvement in terms of memory. Additionally, trivial expansions amount to a large percentage of the total expansions yielding significant time reduction. Furthermore, reduction of runtime is achieved for many values of k. The best runtimes over all the lookaheads is labeled in **bold**. For the 15-puzzle k=4 produced the greatest speedup, achieving a 9-fold reduction of A*'s runtime accompanied by 10-fold reduction in the memory required. AL*(4) also gains more than a factor of 2 reduction in runtime over IDA*. For TopSpin (which has many short cycles), k=1 produced the greatest speedup, achieving a 2-fold reduction of A*'s runtime and an 8-fold reduction of memory. Note that since the TopSpin state space contains many transpositions, IDA* performs poorly on it. Additionally, even a lookahead of zero may save memory, since in AL^* nodes are goal tested when they are generated (unlike text book A* implementation).[3] Generating a goal node sets an upper bound on the goal cost, pruning future nodes with equal or higher cost. In the 15-puzzle with MD the goal node and its parent have the same f-value (moving the last tile to its place), thus no memory reduction is gained with AL*(0).

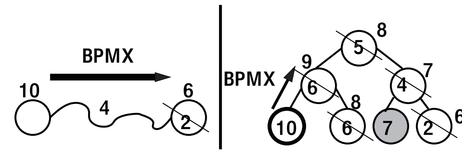

Figure 3: BPMX examples.

Lookahead with Inconsistent Heuristics

As explained above, in AL^* the cost of a child node f_u is updated during the lookahead by propagating the minimum cost found in the lookahead frontier. When inconsistent heuristics are available (and an undirected state space) more propagation can be done.[4]

When inconsistent heuristics are available, admissible heuristic values can propagate between connected nodes x and y. $h(x)$ can be updated to be $max(h(x), h(y)-d(x,y))$. This is shown in Figure 3 (left). The value from the left node (10) is propagated to the right node which now becomes 6 instead of 2. Applying this from parents to children (and further to their descendants) is called *Pathmax* (Mero 1984). Recently it has been shown that this can be also applied in both directions. Values from children can propagate to their parents (and further to their ancestors or to the other children of the parents). This is called *Bidirectional Pathmax* (BPMX) (Felner et al. 2005). An example of BPMX on a search tree is shown in Figure 3 (right). Assuming all edges costs are 1 then the h-value of the left grandchild (10) is propagated up and down the tree increasing values of all its neighborhood (except for gray node, as its value remains 7).

Both directions of BPMX (nodes to their descendants and vice versa) can be incorporated in IDA* very easily. Heuristic values are passed from a node to its descendants when the DFS deepens and can propagate up the tree when the DFS backtracks. A large reduction in the search effort can be obtained when BPMX is implemented on top of IDA* with inconsistent heuristics (Felner et al. 2005). By contrast, in A*, BPMX can only be implemented efficiently between an expanded node v and its immediate generated children. These nodes are at hand when expanding v and are easily manipulated. The maximal heuristic value among the children can be propagated to v by decreasing it by one (assuming a uniform edge cost) and then to the other children by decreasing it by one again. This process is called BPMX(1) (Zhang et al. 2009) because the propagation is only performed one level from v. Further propagation is less efficient because nodes that might be updated may not be available and we need to retrieve them from OPEN or CLOSED. In addition,

[3]This can also be implemented in A* (Ikeda and Imai 1999).

[4]A heuristic h is *inconsistent* if there exist at least two states, x and y such that $(h(x) - h(y)) > dist(x,y)$ where $dist(x,y)$ is the shortest path between x and y.

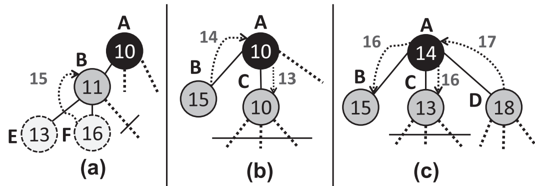

Figure 4: Node expansions using $AL^*(2)$ with BPMX.

(Zhang et al. 2009) showed that this might lead to an exponential blowup and concluded that only BPMX(1) is effective and practical in A*.

One of the contribution of this paper is that in AL*(k) deeper propagation than BPMX(1) can be performed without these problems as BPMX can be can be integrated efficiently along the DFS phase up to the lookahead depth. It is also possible to use BPMX(1) in AL* to further improve the accuracy of the cost of generated child nodes. There are three independent possible locations in AL* where BPMX(1) can be applied. (1) Before the lookahead (before line 4 in Algorithm 2) to avoid performing redundant lookaheads. (2) After the lookahead updated the cost of the generated node (after line 20 in Algorithm 2). Here we propagate the new cost found at a node to its siblings before performing a lookahead from them (and possibly pruning them). (3) After all the lookaheads are performed, to update the cost of all the child nodes according to the best heuristic cost found throughout the lookaheads. The nodes will be inserted to OPEN with this updated cost (after line 23 in Algorithm 2). As each call to BPMX(1) has an overhead, it might not be effective to use all three together. We found out that the best balance is achieved by using only options 2 and 3 for some domains and using all three options for other domains.

Figure 4 provides an example of an expansion cycle of AL*(2) with BPMX. Nodes are annotated with their f-cost. Generated nodes are colored in gray, lookahead nodes are colored in white and expanded in black. A has three children B,C and D. First, B is generated (Figure 4a) and a lookahead is performed. $f(F)$ (16) is propagated with BPMX up to B. Since the lookahead limit ($f(A) + 2 = 12$) has been reached, the lookahead halts, without visiting other children of B. Next, C is generated (Figure 4b) and the cost of B is propagated to it using BPMX(1) (option 2 described above). Since the new cost of C exceeds the lookahead limit ($13 > 12$), a lookahead is not performed from C. Finally, D is generated (Figure 4c), with a cost of 18. This cost is propagated to B and C using option (3) described above. B and C are now inserted to OPEN with cost 16.

Experimental Results

We implemented AL*(k) with BPMX on several domains, but present here only the results of TopSpin. Table 3 presents average results over 100 random instances of TopSpin(16,4). The heuristic used is a random choice between a variety of heuristics that were derived from symmetric (geometrical) lookups on an 8-token PDB. We also added two implementations of BPMX: A* with BPMX(1) and IDA* with BPMX.

Clearly, AL*(k) with BPMX substantially reduces the memory requirements. Additionally, a significant improvement in runtime is achieved. A lookahead of 1 yielded the best runtime, with an improvement factor of more than 2 over A* with BPMX(1) (plus a factor of 11 reduction in memory), and a factor of more than 3 over IDA* with BPMX and simple A*.

Algorithm	Memory	Expanded	DFS	Time
A*	1,015,440	121,154	-	3.60
$A^*_{BPMX(1)}$	636,131	76,880	-	2.97
AL*(0)	621,394	70,706	334,011	2.28
AL*(1)	47,847	6,530	857,844	**1.30**
AL*(2)	7,806	925	1,239,665	1.66
AL*(3)	882	96	1,448,911	1.91
IDA*	-	-	3,697,245	5.58

Table 3: AL^* on TopSpin(16,4) with random heuristic.

Conclusion and Future Work

We introduced an approach to incorporate a DFS-based lookahead into BFS algorithms, and in particular to BRFS and A*. We also showed BPMX propagation in AL* is implemented better than with A* when inconsistent heuristics are used. Experimental results showed reduction in both memory and in time for a variety of domains. Future research will continue in a number of directions. First, we would like to mathematically predict the optimal lookahead given the different attributes of a domain. Second, the lookahead depth can be changed dynamically based on dynamic changes of the attributes of the domain.

Acknowledgments

This research was supported by the Israeli Science Foundation grant no. 305/09 and by the iCORE and NSERC grants.

References

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and de Sarkar, S. C. 1989. Heuristic search in restricted memory. *Artif. Intell.* 41(2): 197–221.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI*, 103–108.

Ikeda, T., and Imai, H. 1999. Enhanced A algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theor. Comput. Sci.* 210(2): 341–374.

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

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

Mero, L. 1984. A heuristic search algorithm with modifiable estimate. *Artif. Intell.* 23(1): 13–27.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.* 16(7): 701–710.

Russell, S. J. 1992. Efficient memory-bounded search methods. *Proc of ECAI-92*.

Sen, A., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of IJCAI-89*, 297–302.

Sun, X.; Yeoh, W.; Chen, P.-A.; and Koenig, S. 2009. Simple optimization techniques for A*-based search. In *AAMAS (2)*, 931–936.

Zhang, Z.; Sturtevant, N. R.; Holte, R. C.; Schaeffer, J.; and Felner, A. 2009. A* search with inconsistent heuristics. In *IJCAI*, 634–639.