

Best-First Search with Lookaheads

Tamar Kulberis

Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel
kulberis@bgu.ac.il

Roni Stern

Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel
roni.stern@gmail.com

Ariel Felner

Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel
felner@bgu.ac.il

Abstract

Best-First Search (BFS) is a classic general search technique. It maintains an open-list of generated nodes and expands the least-cost node from it while adding its unvisited neighbors to the open-list. The main limitation of BFS is that it stores all the states it visits in memory. BFS is composed by a number of primitive steps and can be implemented in many ways. In this paper we deepen into implementation details of BFS by analyzing the standard primitive steps and their execution order. In particular, we study the roles of the *duplicate detection* and of the *goal test* primitives. Based on this study we introduce a new BFS variation called BFS with Lookahead (BFSL). The basic idea of BFSL is to perform limited DFS lookaheads from the frontier of the BSF (open-list). We show that this algorithm requires significantly less memory. In addition, a time speedup is also achieved when choosing the lookahead depth correctly. Experimental results on several domains demonstrate the benefits of all our ideas.

Introduction

Best-first search (BFS) is a well-known 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*) from OPEN. When a node is expanded it is moved from OPEN to CLOSED, and its children are generated and added to OPEN. The search terminates when a goal node is chosen for expansion, or when OPEN is empty. Figure 1 shows the pseudo code of a general BFS of the common AI text book (Russell and Norvig 2005).

Many known algorithms are special cases of BFS differing only in their cost function. If the cost of a node is its depth in the tree, then BFS becomes breadth-first search (denoted here as BRFS), expanding all nodes at a given depth before any nodes at any greater depth. If the edges in the graph have different costs, then taking $g(n)$, the sum of the edge costs from the start to node n as the cost function, yields Dijkstra's algorithm. If the cost is $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimation of the cost from node n to a goal, then BFS becomes the A* algorithm. BFS is complete, and in the case of A* with an admissible heuristic (i.e., a heuristic that never overesti-

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

```
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
  if EMPTY?(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST[problem] applied to STATE[node] succeeds
    then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

function EXPAND(*node*, *problem*) **returns** a set of nodes

```
successors ← the empty set
for each (action, result) in SUCCESSOR-FN[problem](STATE[node]) do
  s ← a new NODE
  STATE[s] ← result
  PARENT-NODE[s] ← node
  ACTION[s] ← action
  PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
  DEPTH[s] ← DEPTH[node] + 1
  add s to successors
return successors
```

Figure 1: Standard BFS from (Russell and Norvig 2005)

mates the actual cost from node n) returns optimal solutions and is optimally effective (Dechter and Pearl 1985).

The main drawback of BFS is its memory requirements. BFS stores in memory all the *open* and *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.

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, there is no guarantee that the solution is optimal. The memory limitation of BFS has been addressed in the past and several algorithms that generate nodes in a best-first fashion while activating different versions of DFS have been developed. The most common algorithm is Iterative-deepening A* (IDA*) (Korf 1985) which performs a series of DFS calls from the root in an increasing order of costs. Another known linear-space algorithm is Recursive best-first search (RBFS) (Korf 1993). It expands new nodes in a best-first order even when the cost

Algorithm 1: Basic BFS node expansion cycle

Input: n , the best node in OPEN

```
0.1 if goalTest( $n$ )=True then
0.2 | halt ; /* Goal was found */
0.3 insert  $n$  to CLOSED
0.4 foreach state operator  $op$  do
0.5 |  $child \leftarrow$  generateNode( $op, n$ )
0.6 | if duplicateDetection( $child$ )=False then
0.7 | | insert  $child$  to OPEN
0.8 | else
0.9 | | update cost function of  $duplicate$  (if required)
0.10 | end
0.11 end
```

function is *nonmonotonic*. Since these algorithms are DFS oriented their efficiency significantly deteriorates in state spaces that contain many cycles or a very large distribution of heuristic values.

Other attempts to search in a best-first order but with a limited amount of memory include MREC (Sen and Bagchi 1989), MA* (Chakrabarti et al. 1989) and SMA* (Russell 1992). While these algorithms may generate fewer nodes than IDA* they run slower in practice because of memory maintenance overhead (see (Korf 1993)).

In this paper we explore approaches to enhance BFS performance. First, we study the implementation details of BFS. While the basic mechanism of BFS is well-known, we show that standard textbook implementation can be improved. BFS includes many different primitive steps such as *duplicate detection*, *goal test*, *insertion* and *deletion* from different data structures (e.g., OPEN and CLOSED). We study different implementations and discuss the pros and cons of each with regards to the time and memory needs.

Second, based on this study, we introduce a new algorithm called *BFS with Lookahead* (BFSL). The basic idea is to use the general schema of BFS but to perform limited DFS lookaheads from nodes when they are expanded. BFSL 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, we show that when correctly choosing the lookahead depth, we can control the problem of duplicates and cycles which occur in DFS and thus significant time speedup can be also obtained in many cases. Experimental results on several domains demonstrate the benefits of all our ideas.

The next sections discuss several primitive steps of BFS and study some aspects of their implementation details. Then the new algorithm, BFS with lookahead (BFSL), will be introduced.

Duplicate Detection

Algorithm 1 presents the basic steps of the expansion cycle of the least cost node from OPEN. A number of steps are performed on the node just extracted from OPEN. First, a goal test is performed. Then, all its children are generated. For each child c , OPEN and CLOSED are checked to see

<i>DuplicateDetection</i>	<i>CDD</i>	<i>UDD</i>
False	COPY CHANGE DD? INSERT -	CHANGE DD? COPY INSERT UNDO
True	COPY CHANGE DD?	CHANGE DD? UNDO

Table 1: CDD Vs. UDD

whether they already contain duplicates of c . This step is called *duplicate detection* (DD). If no duplicate was found, c is inserted to OPEN. In order to perform an efficient DD both OPEN and CLOSED are usually maintained in a hash table. In addition, OPEN should also be maintained as a priority queue (usually implemented intermixed with the hash table) in order to allow the extraction of the least-cost node.

Change and Undo Vs. Copy and Discard

The generation of a child usually involves two atomic steps.

- **COPY:** copying the parent node into a new data structure.
- **CHANGE:** applying the selected operator (e.g. a moving of a tile in the tile puzzle) on the new child, causing a change in its description.

The relative time required for each of these steps is domain/implementation dependent. However, it is possible to evaluate a node without completely generating it. This can be done by modifying its parent node, evaluating it, and undoing the modification. This observation allows the two following implementation variants of generating a node (lines 4-10) in the basic BFS described in Algorithm 1.

- **Copy and Discard Duplicate Detection (CDD):** In this version a node is generated by first copying the parent and then applying the operator. The DD operation is performed on the newly generated node.
- **Change and Undo Duplicate Detection (UDD):** Here, we first CHANGE the parent and perform the DD test. Only if DD returns FALSE we COPY this node into a new state item and insert it to OPEN. Then, we perform an UNDO operation to get back the parent node.

Table 1 shows the different operations taken by both CDD and UDD based on the result of the DD test. The following primitive steps are used in the table. **Duplicate-detection (DD)** is the matching of a node against OPEN and CLOSED. This operation returns TRUE or FALSE. An **INSERT** operation inserts the new child to OPEN and an **UNDO** operation is the reverse of the CHANGE operation.

Choosing the more efficient variant depends on the domain and the state representation. For domains with a large number of duplicate nodes and a costly COPY operation, UDD will probably perform faster, while for domains with relatively small rate of duplicate nodes and cheap COPY operation (e.g. with a compact state representation), CDD might be preferable. Formally, let P_{dd} be the probability

Goal test	Representation	UDD	CDD
At expansion	Regular	24.5 sec	26.8 sec
At expansion	Packed	51.0 sec	40.1 sec
At generation (early)	Regular	14.1 sec	15.6 sec
At generation (early)	Packed	31.0 sec	24.4 sec

Table 2: CDD and UDD with different representations

that a duplicate node was found (DD returns True), and let C_x be the constant cost of performing action x . Since both CDD and UDD perform exactly the same expansions, it is enough to compare the expected constant time of their expansion cycle as follows.

$$T(CDD) = P_{dd} \times (C_{change} + C_{dd} + C_{undo}) + (1 - P_{dd}) \times (C_{change} + C_{dd} + C_{copy} + C_{insert} + C_{undo})$$

$$T(UDD) = P_{dd} \times (C_{copy} + C_{change} + C_{dd}) + (1 - P_{dd}) \times (C_{copy} + C_{change} + C_{dd} + C_{insert})$$

By comparing these two equations and eliminating common elements, we conclude that the relative speed of CDD Vs. UDD depends on the duplicate node probability (P_{dd}) and the constant of C_{copy} and C_{undo} . If $P_{dd} \times C_{copy} > C_{undo}$ than UDD will be faster. Otherwise CDD will be faster. Note that P_{dd} is an attribute of the problem domain, while C_{copy} and C_{undo} are attributes of state representation.

Experimental results: We implemented both CDD and UDD on two state representations of the 15 puzzle. In the first representation (*explicit representation*), each state is represented as an array of size 16 (=number of tiles). In the second representation (*packed representation*) the explicit state is *packed* and saved in 2 words. Performing CHANGE (or UNDO) operation requires unpacking the node to the explicit representation while COPY operation simply requires copying 2 words. Table 2 presents the results of these experiments (first two lines). Each value in the table is the average runtime of using BRFS over a set of 50 instances of the 15 puzzle located 22 moves away from the goal. All the experiments in this paper were run on 2GHz Core2Duo PC with 2GB of memory.

For the explicit representation, UDD outperforms CDD while for the packed representation, CDD is more efficient. This is explained by the UNDO and COPY steps in each representation. In the explicit representation, an UNDO step contains a change in two cells of the state array (switching between the blank and a tile) while a COPY step requires copying all the 16 cells. By contrast, in the packed representation an UNDO operation requires packing back the parent state. Thus, performing UNDO for all the generated nodes is more costly than copying some of the generated node. The last two lines of this table will be treated later.

Late Duplicate Detection

In standard BFS implementation, DD is performed as soon as a node is generated. Korf in his seminal work on disk-based graph search summarized in (Korf 2008) argued that if DD requires searching external memory it will be highly inefficient to perform it at every node generation. He suggested the *delayed duplicate detection* mechanism (DDD) for disk-based BFS searches. BFS with DDD stores OPEN

Algorithm 2: Pseudo code for late duplicate detection

Input: n , the *best* node in OPEN

```

1.1 if duplicateDetection( $n$ )=True then
1.2 |   return ; /* Node already in CLOSE */
1.3 if goalTest( $n$ )=True then
1.4 |   halt ; /* Goal was found */
1.5   insert  $n$  to closed list
1.6 foreach state operator op do
1.7 |    $child \leftarrow$  generateNode( $op, n$ )
1.8 |   insert  $child$  to open list
1.9 end

```

and CLOSE intermixed on a single list. During the expansion step, all nodes with a specific cost are expanded, and their successors are added to the end of the node list without checking for duplicates. The nodes are then sorted by their state representation, and thus duplicate nodes representing the same state have adjacent locations. By linear scanning the sorted list periodically (e.g., after expanding an entire level in BRFS), all duplicate nodes are removed, while saving a single copy of each node with the minimum cost. (Korf 2008) also describes a possible way of implementing this mechanism in an in-memory BFS algorithm.

We propose an alternative way to delay the DD operation in an in-memory implementation. Our method is only a slight modification of the regular implementation of BFS. Instead of performing DD on newly generated nodes, the DD can be performed later when a node is chosen for expansion. To distinguish from Korf's DDD we call this variation *late duplicate detection* (LDD). Algorithm 2 presents the pseudo code for BFS with LDD. Every generated child is inserted to OPEN and nodes are checked for duplicates only when expanded.

The expected time benefit of performing LDD is twofold. First, the constant time per node generation decreases without the need to perform a costly DD which involves calculating the hash value as well as costly pointer chasing in the collision list. Since many nodes are generated without being expanded (all the nodes in OPEN when the goal node is found), this is significant time saving. Second, the most important advantage of LDD is that the DD test is performed only on CLOSED, instead of both OPEN and CLOSED. Therefore, OPEN need not be stored in a hash table and a simple implementation of a priority queue (e.g. a heap) is sufficient. As a consequence insertion and extraction from OPEN should be faster. Furthermore, memory can be saved too as OPEN nodes no longer need to be stored in the hash table which may require additional pointer for each item. All this comes at the cost that duplicate versions of a node can exist in OPEN, thus wasting memory. But, if the number of small cycles and duplicates is small, LDD might reduce both memory and time.

Experimental results The first two lines of table 3 below present the results of running standard BRFS and BRFS with LDD. Each value in the table is the average over the same set of 50 instances of the 15 puzzle. The *expanded* column shows the number of nodes expanded during the search.

The *open* column shows the number of nodes inserted into OPEN, and the *time* column shows the runtime in seconds. LDD inserted 10% more nodes to CLOSED and 20% more nodes to OPEN. However, since OPEN nodes needed less constant memory the total memory was reduced as well. The number of nodes stored in the hash table is written in **bold**: only 11,164,851 for LDD Vs. 19,473,242 for standard DD. Since each item in the hash table requires its own pointer, this dominates the additional duplicate nodes that were inserted to OPEN with LDD. Moreover, due to light constant time per node LDD improved the total runtime from 16.62 to 13.90 seconds.

Early Goal Test During Child Generation

Goal test is an important primitive step in any search algorithm used to identify a goal node. When such node is identified, the search usually halts. Textbooks (such as the one provided above) usually teach that a *goal test* in BFS is performed only when a node is chosen for expansion and is extracted from OPEN. This is valid for all variations of BFS (including BRFS). However, good implementers of BRFS usually recognize that the search can stop earlier if a goal test is performed when a node is generated. We call this *early goal test*. This reduces both time and memory because the entire round in which this goal node goes through OPEN (implemented as a FIFO queue in BRFS) until it reaches its front and extracted can be omitted. This saves the processing of a full level of the tree. For example, we have found that this improvement alone reduces memory consumption of BRFS on the 15-puzzle by a factor of 2.

Unlike BRFS, for the general case of BFS (e.g., A*) generating a goal does not guarantee that the optimal solution has been found because a goal with a lower cost can be identified later in the process. Thus, implementers usually follow the textbooks and only halt the search while a goal node is chosen for expansion. However, as recognized by some researches (Hansen and Zhou 2007)(Likhachev, Gordon, and Thrun 2003)(Ikeda and Imai 1999)(Zhou and Hansen 2003) early goal test is valid for general BFS too.

In the case of BFS, early goal test is performed on generated nodes and the cost of the best solution found so far is used as an upper bound, UB, (initialized to ∞). Once the goal has been generated, the role of the following expansions is to verify that no better solution exists. This is verified when the cost of the best node in OPEN is *greater than or equal to* UB. So only nodes with cost *less than* UB should be expanded. Thus, as soon as the goal node has been generated with a cost of c , we can immediately delete from OPEN all nodes with costs *greater than or equal to* c . Also, any new generated node with such costs can be pruned immediately as it cannot lead to a better solution. Significant amount of time and memory can be saved this way.

Algorithm 3 presents the pseudo code of a node expansion cycle with the early goal test enhancement. We believe that this version of performing goal test when generating a node should become the standard way for describing BFS and BRFS and hope that textbooks will adopt it. This idea of early goal test is generalized later in our new BFS with lookahead algorithm.

Algorithm 3: Expansion cycle of BFS with early goal test

```

Input:  $v$ , the best node in OPEN
Input: UB, an upper bound (initialized with  $\infty$ )
2.1 if  $cost(v) \geq UB$  then
2.2 | halt; /* Optimality verified */
2.3 insert  $v$  to CLOSED
2.4 foreach operator  $op$  do
2.5 |  $child \leftarrow generateNode(op, v)$ 
2.6 | if  $cost(child) \geq UB$  then
2.7 | | continue; /* Prune the node */
2.8 | if  $goalTest(child)=True$  then
2.9 | |  $UB=\min(UB, cost(child))$ 
2.10 | | Delete nodes with  $cost \geq UB$  from OPEN.
2.11 | if  $duplicateDetection(child)=False$  then
2.12 | | insert  $child$  to OPEN
2.13 | else
2.14 | | update cost function of  $child$  (if required)
2.15 | end
2.16 end

```

Algorithm 4: Expansion cycle of BRFSL(k)

```

Input:  $n$ , the best node in OPEN
3.1 foreach state operator  $op$  do
3.2 |  $child \leftarrow generateNode(op, n)$ 
3.3 | if  $DFS-with-GoalTest(child, k)=True$  then
3.4 | | halt; /* Goal was found */
3.5 | if  $duplicateDetection(child)=False$  then
3.6 | | insert  $child$  to OPEN
3.7 | else
3.8 | | update cost function of  $duplicate$  (if required)
3.9 | end
3.10 | insert  $n$  to CLOSED
3.11 end

```

Experimental results: The last two lines of table 2 show UDD and CDD results for both representation methods when running BRFS with early goal test. It is easy to see that performing a goal test during node generation is much more efficient, saving approximately 36% for the explicit state representation and about 40% for the packed state representation for both UDD and CDD. Results for general BFS with early goal test will be presented below as special case of our new algorithm BFS with lookahead.

Best-First Search with Lookahead

Based on the different ideas presented above we now turn to describe our new algorithm, *BFS with lookahead* (BFSL) which generalizes these ideas and introduces a novel combination of limited DFS and BFS. For simplicity we begin by describing it on breadth-first search and then move to the more general case of best-first search.

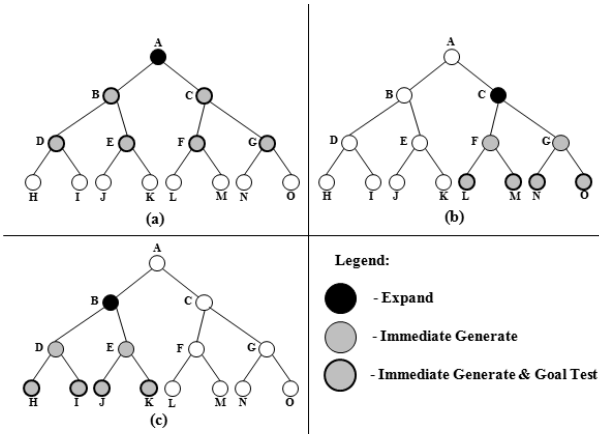


Figure 2: Example of BRFS(2)

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 iteration, 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 . Goal test is only performed on nodes at depth k . 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 immediate successors are added to OPEN. Algorithm 4 presents the pseudo code for BRFSL. A limited DFS from *child* is performed. If a goal is found during this limited DFS, then *lookaheadGoalTest* return *True*.

An exception 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 (or a limited BRFS) 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. Note that ordinary BRFS is the special case of BRFSL(0), while BRFS with early goal test described above is the special case of BRFSL(1).

Three expansion cycles of BRFSL(2) are illustrated in Figure 2. 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. Before generating its immediate successors (D and E), a lookahead to depth 2 is performed, where only nodes at the deepest lookahead level are tested for goal (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 the algorithm 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 earlier expansions, when a goal node is found at depth d it is optimal.

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 unique successors (after applying DD). It is important to note that $b_e \leq b$, due to cycles 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 space complexity of BRFSL(k) (in the worst case, where the goal node is found below the last node at depth $d - k$) is

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

Clearly this requires less space than standard BRFS (BRFSL(0)) which requires $O(b_e^d)$ memory.¹

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 $1 + b_e + b_e^2 + b_e^3 + \dots + b_e^{d-k}$ such nodes. *Visited* nodes are nodes that are visited during the lookahead DFS calls. A single DFS lookahead search to depth k visits $1 + b + b^2 + b^3 + \dots + b^k = O(b^k)$ nodes. We perform such lookaheads for all expanded nodes and thus the total number of nodes that are visited during all DFS lookaheads is $(1 + b_e + b_e^2 + b_e^3 + \dots + b_e^{d-k}) \times (1 + b + b^2 + b^3 + \dots + b^k)$. This amounts to $O(b_e^{d-k} \times b^k)$ which is larger than the number of nodes visited by ordinary BRFS ($= 1 + b_e + b_e^2 + b_e^3 + \dots + b_e^d = O(b_e^d)$) 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 cycles, 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 use b_e .

However, in practice BRFSL(k) may run faster than BRFS for a number of reasons. First as in BRFS with early goal test, we stop the search when a goal is reached during a lookahead. Second, the lookahead nodes can be generated by only performing CHANGE operations on the expanded node. This avoids the need to COPY an entire state for every node in the lookahead, yielding an improved runtime (similar to UDD above). 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 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 ordinary BRFS (BRFSL(0)) and BRFS with early goal test (BRFSL(1)).

¹Strictly speaking, b_e is not constant and may vary during the depth of the search. However it is always less than or equal to b , and thus our analysis is valid.

Fifteen Puzzle				
k	Expanded	Opened	DFS	Time
0	10,373,537	19,473,242	0	16.62
0+LDD	11,164,851	23,449,495	0	13.90
1	5,978,496	11,305,305	12,566,619	9.56
1+LDD	5,902,723	18,310,493	12,566,619	8.94
2	3,138,531	5,978,495	20,674,663	6.10
2+LDD	3,098,076	9,616,294	20,674,663	6.40
3	1,634,463	3,138,530	26,483,929	4.80
4	845,867	1,634,462	31,085,133	4.26
5	434,590	845,866	34,966,998	4.25
6	222,132	434,589	38,537,864	4.31
7	112,758	222,131	41,900,800	4.60
8	56,961	112,757	45,243,397	4.78
9	28,568	56,960	48,437,217	5.21
10	14,258	28,567	51,578,154	5.37
11	7,050	14,257	54,304,459	5.78
12	3,467	7,049	56,902,134	5.92
13	1,682	3,466	58,866,868	6.25
14	808	1,681	60,560,268	6.29
15	385	807	61,653,712	6.58
16	182	384	62,489,184	6.48
17	86	181	62,764,526	6.69
18	40	85	62,911,950	6.58
19	18	39	63,225,908	6.55
20	8	17	63,894,484	6.35
21	3	7	65,318,508	6.00
(12,4) TopSpin Puzzle				
0	1,845,383	12,575,891	0	17.35
1	258,032	1,854,382	3,096,386	3.31
2	34,887	258,031	5,442,500	2.99
3	4,592	34,886	8,651,828	4.5
4	583	5,591	13,207,502	6.68

Table 3: Results of BRFSL with various lookaheads

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 the constant time per node is smaller. The optimal value for k is domain dependent and is strongly related to the rate of duplicates and cycles 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 3 presents the results averaged on our 50 depth-22 instances. The k column shows the lookahead depth (where $k = 0$ is ordinary BRFS and $k = 1$ is simply BRFS with early goal test). The *Expanded* column shows the number of nodes expanded during the search, the *Opened* column shows the number of nodes inserted into OPEN, 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 (= number of nodes in the hash table 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 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 TopSpin(12,4) at depth 14. They are presented 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 smaller cycles.

The table also shows results for combining LDD and BRFSL for small k values. With LDD the search visited more nodes but a smaller number of nodes were stored in the hash table (labeled in **bold**). This reduced the overall memory needs. In addition LDD yielded a speedup in the runtime for $k = 0$ and $k = 1$.

A* with Lookahead

We now generalize this concept to best-first search with lookahead, using A* with lookahead (AL*) as an example.

Trivial Lookahead

In BRFS the cost function is the depth of the node ($f = d$). Thus, when expanding a node with cost x we always generate nodes with cost $x + 1$. By contrast, in A* the cost function is ($f = g + h$) and if h decreases, a generated node may have the same cost as its parent. In this case, the generated node is entered to OPEN, but can then be immediately expanded. Inserting to OPEN is a costly operation as priority queues times are generally $O(\log(n))$. Thus, we suggest a simple enhancement to A* where generated nodes that have the same cost as their parents (just expanded), are immediately expanded too. This is done as follows. Assume that node v with cost c is extracted from OPEN. We now perform a DFS rooted at v . Successors with a cost of c are immediately goal tested and added to CLOSED. When successors with cost greater than c are encountered, the DFS backtracks and these nodes are generated and added to OPEN. It is important to note that this enhancement is only valid if the cost function is *monotonic increasing* along any branch. In domains where many successors share the same cost function, this enhancement is expected to substantially reduce execution time, as we bypass OPEN for many nodes in comparison to the classic A*. Note that this enhancement makes BFS more like BRFS, where each expanded node adds nodes with the next largest cost. We denote this enhancement as *trivial lookahead*.

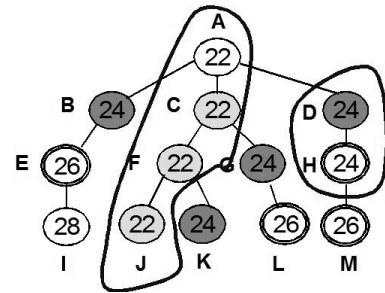


Figure 3: Example of an AL* iteration.

For example, consider Figure 3. The search starts at node A and the goal node is I . Using trivial lookahead, nodes B , F and J will be immediately expanded when expanding A and inserted straight to CLOSED, saving the cost of inserting and extracting them from OPEN. Similarly, H will

be immediately expanded when expanding D . Each cluster of nodes that are expanded at the same time is circled.

A* with Arbitrary Lookahead

Algorithm 5: Expansion cycle of A* with lookahead

```

Input:  $v$ , the best node in OPEN
Input:  $UB$ , an upper bound, initialized with  $\infty$ 
4.1 if  $cost(v) \geq UB$  then
4.2 |   halt /* Optimality verified */
4.3
4.4 insert  $v$  to CLOSED
4.5 foreach operator  $op$  do
4.6 |    $child \leftarrow generateNode(op, v)$ 
4.7 |   if  $cost(child) \geq UB$  then
4.8 | |   continue /* Prune the node */
4.9 |   if  $goalTest(child)=True$  then
4.10 | |    $UB=cost(child)$ 
4.11 | |   Delete nodes with  $cost \geq UB$  from Open
4.12 |   if  $cost(child)=cost(v)$  then
4.13 | |    $expand(child)$  /* Trivial lookahead */
4.14 |   else
4.15 | |   if  $duplicateDetection(child)=False$  then
4.16 | | |   insert  $child$  to open list
4.17 | |   else
4.18 | | |   update cost function of  $child$  (if required)
4.19 | |   end
4.20 |    $maxDepth \leftarrow \min(UB, cost(v) + k)$ 
4.21 |   if  $cost(child) \geq maxDepth$  then
4.22 | |    $lookahead(child, maxDepth, UB)$ 
4.23 | |   /*  $UB$  is possibly updated */
4.24 |   end
4.25 end
4.26 end

```

Procedure $lookahead(v, maxDepth, UB)$

```

5.1 foreach operator  $op$  do
5.2 |    $child \leftarrow generateNode(op, v)$ 
5.3 |   if  $cost(child) \geq maxDepth$  then
5.4 | |   return
5.5 |   if  $goalTest(child)=True$  then
5.6 | |    $UB=cost(child)$ 
5.7 |   else
5.8 | |    $lookahead(child, maxDepth, UB)$ 

```

Extending trivial lookahead to arbitrary lookahead of k (denoted $AL^*(k)$) is done as follows. Assume that the cost of the expanded node is c . We perform trivial lookahead, immediately expanding nodes with costs c . When a successor with a cost *greater than* c is reached, it is inserted to OPEN but a limited DFS is first activated from that node. Each node encountered during this DFS is goal tested. If a goal has been found with cost smaller than the currently best known solution UB , then UB is updated. At this stage it is possible to go over all the nodes in OPEN and remove nodes with cost *greater than or equal to* UB .² In addition, as in

²While this option saves memory, it slows the total execution

early goal test, nodes with cost *greater than or equal to* UB are discarded. The search halts when the optimal goal cost has been verified, i.e., when the cost of a node chosen for expansion is equal to UB . Pseudo code for $AL^*(k)$ is presented in Algorithm 5. Note that when $k = 0$, the algorithm describes the trivial lookahead mentioned above.

Figure 3 illustrates an expansion cycle of $AL^*(4)$, 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 light gray 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 smaller than 26 ($22+4$) are visited and goal tested. Thus the nodes visited during the lookahead are E, L, H and M , marked with a doubled circle.

In many domains the costs of nodes increase in a fixed amount. For example, in the tile puzzle, nodes with cost of x can have children with costs of either x or $x + 2$. In such cases $AL^*(k)$ is very similar to $BRFSL(k)$. When nodes with a certain cost are expanded, nodes with the next cost are inserted to OPEN.

Experimental Results

k	Expanded	Trivial	Opened	DFS	Time
Fifteen Puzzle: 7-8 PDB					
A*	21,004	0	41,676	0	0.30
A*e	21,004	0	41,676	0	0.30
0	21,004	17,202	25,474	0	0.14
2	18,245	15,181	4,602	63,163	0.24
4	2,801	2,365	848	136,988	0.30
6	399	336	141	180,275	0.38
8	55	47	17	202,664	0.42
10	7	6	1	233,587	0.48
Fifteen Puzzle: Manhattan Distance					
A*	2,321,155	0	4,200,948	0	14.11
A*e	2,321,155	0	4,200,946	0	14.11
0	2,321,155	1,530,457	2,670,491	0	6.13
2	2,119,428	1,585,501	702,325	7,464,985	4.06
4	492,066	372,596	190,196	18,449,760	2.32
6	108,911	83,231	48,833	27,344,284	2.15
8	23,174	17,804	11,795	35,824,818	2.71
10	4,790	3,686	2,610	43,442,763	3.12
(16,4) Top Spin					
A*	662,256	0	5,109,005	0	45.38
A*e	662,256	0	2,226,886	0	33.10
0	662,256	419,813	1,807,073	0	29.80
1	524,022	365,178	1,022,571	17,194,709	34.06
2	68,133	48,615	194,941	26,025,625	24.56
3	8,095	5,935	31,728	38,403,901	33.00
4	1,349	971	5,051	59,933,594	51.27

Table 4: AL^* with on the puzzles

We implemented AL^* on the 15-puzzle with Manhattan distance and with the 7-8 additive PDB heuristic (Korf and Felner 2002) and on the (16,4)-TopSpin with a 9-token PDB time. We did not implement it in our experiments.

heuristic. Table 4 presents the results averaged over 100 random instances on both puzzles. The k column indicates the threshold of the lookahead, where 0 denotes the trivial lookahead. *Expanded* counts the total number of nodes that entered CLOSED while *Trivial* counts the number of nodes that entered CLOSED while bypassing OPEN (trivial lookahead). The rest of the columns are similar to previous tables. Trivial expansions amount to more than 70% of the total expansions in both puzzles yielding both memory and time savings.

In both domains using AL* with larger lookaheads yields substantial improvement in terms of memory. For example, in the 15 puzzle, using a lookahead of 2 reduces the number of stored nodes³ by a factor of 3, and in TopSpin using a lookahead of 3 reduces the number stored nodes by two orders of magnitude. Additionally, the results show that for many k values substantial reductions of time can be achieved. The best runtimes over all the lookaheads is denoted in *bold*. The optimal time was achieved using lookahead of 2 for TopSpin (a $\times 2$ reduction over A*). For the 15 puzzle the optimal time was for lookahead 0 (trivial) with the 7-8 PDB ($\times 2$ reduction) and 6 for the 15 puzzle with Manhattan distance ($\times 7$). The explanation for this discrepancy is the constant time of the heuristic calculation. The 7-8 PDB is very large and consulting it probably results in poor cache performance. Thus, the heuristic lookups in the DFS phase consume much time. By contrast, Manhattan distance is very fast to compute and the DFS phase is very fast.

The A*e line refers to A* with early goal test. It is interesting to note that A*e is identical to A* in the tile puzzle. The reason is as follows. The goal state has the blank in the upper left corner and has only two neighbors, one of them is its parent in the search tree. It can be shown that for the PDB used, the heuristic only decreases by one when the blank enters its goal state and thus its f -value is the same as its parent. In such cases, early goal test is meaningless as this node would be expanded right away. For TopSpin, the goal can have larger f -value and many node generations can be saved (as can be seen in the *Opened* column).

Conclusion and Future Work

We have shown a number of ways to enhance BFS and BRFS in order to better utilize the memory and time. Additionally, we have introduced a novel approach to incorporate a DFS-based lookahead into BFS algorithms, and in particular to BRFS and A*. Experimental results supported our direction. This research is in progress and the following directions are currently taken.

Combining AL* with and BMPX

Bidirectional pathmax (BPMX) is a method that propagates heuristic values in any possible direction when inconsistent heuristics are used (Felner et al. 2005; Zahavi et al. 2007). BPMX is easily implemented with IDA* or with any other DFS search as values can easily be propagated between nodes and their neighbors during the DFS. By contrast, (Zhang et al. 2009) showed that applying BPMX in A* is much more

problematic and only a limited version of BPMX proved useful within the context of A*. However, when adding DFS lookaheads we can once again use BPMX in the regular DFS form and the potential gain is very large. Initial results show promise but we leave this for a future discussion.⁴

Predicting the Optimal Lookahead

We have shown that by carefully choosing a lookahead depth, both AL* and BRFSL can achieve substantial runtime speedups. However, we have not yet developed a technique for calculating the optimal lookahead. Future work will include analysis of the effects of the lookahead on the runtime and development of optimal lookahead prediction technique. Additionally, we also intend to research the option of a variable depth lookahead. This means learning the domain characteristics throughout the search and adapting the lookahead depth accordingly.

References

- Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and de Sarkar, S. C. 1989. Heuristic search in restricted memory. *Artificial Intelligence* 41(2):197–221.
- Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery* 32(3):505–536.
- Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI-05*, 103–108.
- Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 28:267–297.
- 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. *Theoretical Computer Science* 210(2):341–374.
- Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9–22.
- Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.
- Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.
- Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)* 55(6):1–40.
- Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *NIPS-03*.
- Russell, S. J., and Norvig, P. 2005. *Artificial Intelligence: A Modern Approach. 2nd Edition*. Pearson Education.
- Russell, S. J. 1992. Efficient memory-bounded search methods. *ECAI-92*.
- Sen, A., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI-89*, 297–302.
- Zahavi, U.; Felner, A.; Schaeffer, J.; and Sturtevant, N. R. 2007. Inconsistent heuristics. In *AAAI*, 1211–1216.
- Zhang, Z.; Schaeffer, J.; Sturtevant, N. R.; Holte, R.; and Felner, A. 2009. A* search with inconsistent heuristics. In *IJCAI-09*, to appear.
- Zhou, R., and Hansen, E. A. 2003. Sparse-memory graph search. In *IJCAI-03*, 1259–1268.

³They are the sum of the *Trivial* and *Opened* columns.

⁴We thank Robert Holte for bringing this idea to our attention.