KBFS: K-Best-First Search

Ariel Felner
Dept. of Computer Science
Bar-Ilan University,
Ramat-Gan, 52900 Israel
felner@cs.biu.ac.il

Sarit Kraus
Dept. of Computer Science
Bar-Ilan University,
Ramat-Gan, 52900 Israel
sarit@cs.biu.ac.il

Richard E. Korf
Dept. of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

We introduce a new algorithm, K-best-first search (KBFS), which is a generalization of the well known best-first search. In KBFS, each iteration simultaneously expands the K best nodes from the open-list (rather than just the best as in BFS). We claim that KBFS outperforms BFS in domains where the heuristic function has large errors in estimation of the real distance to the goal state or does not predict dead-ends in the search tree. We present empirical results that confirm this claim and show that KBFS outperforms BFS by a factor of 15 on random trees with dead-ends, and by a factor of 2 and 7 on the Fifteen and Twenty-Four tile puzzles, respectively. KBFS also finds better solutions than BFS and hill-climbing for the number partitioning problem. KBFS is only appropriate for finding approximate solutions with inadmissible heuristic functions.

1 Introduction and overview

Heuristic search is a general problem-solving mechanism in artificial intelligence. The search takes place in a *problem-space graph*. Nodes represent states of the problem and edges represent legal moves. A *problem instance* consists of a problem space together with an initial state and a set of goal states. A solution is a path from the initial state to a goal state. The cost of a solution is the sum of the costs of its edges. A solution is said to be optimal if it is a lowest-cost path from the initial state to a goal state, otherwise it is suboptimal. The term *generating* a node refers to creating a data structure representing the node, while *expanding* a node means to generate all of its children.

1.1 Best-first search

Best-first search (BFS) is a well known and common heuristic search algorithm. It keeps a *closed list* of nodes that have been expanded, and an *open list* of nodes that have been generated but not yet expanded. At each cycle of the algorithm, it expands the most promising node (the *best*) on the open list. When a node is expanded it is moved from the open list to the closed list, and its children are generated and added to the open list. The search terminates when a goal node is chosen for expansion, or when the open list is empty.

Special cases of best-first search include breadth-first search, Dijkstra's single-source shortest-path algorithm [4], and the A* algorithm [8], differing only in their cost functions $f(n)$. If the cost of a node is its depth in the tree, then best-first search becomes breadth-first search, 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 best-first search becomes the A* algorithm. If $h(n)$ is *admissible*, i.e., never overestimates the actual cost from node n to a goal, then A* is guaranteed to return an optimal solution, if one exists.

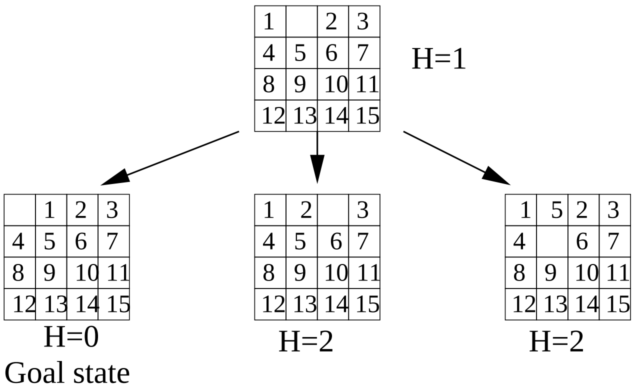

Figure 1: The Fifteen Puzzle

A famous search problem is the sliding-tile puzzle. One example consists of a 4×4 array, with 15 square tiles numbered 1..15 and one blank or empty position. The object of the puzzle is to move the tiles to the goal state in which the tiles are in particular positions. The tiles that are situated next to the blank position can move into the blank space, leaving their former position blank. Figure 1 shows the Fifteen Puzzle. A common heuristic function for the tile puzzle is the *Manhattan distance* heuristic, which is the sum of the number of horizontal and vertical positions that each tile is displaced from its goal position. This heuristic is a lower bound on the actual solution cost, since each tile must move at least its Manhattan distance, and each move moves only one tile.

1.2 Linear space algorithms

A* is complete, returns optimal solutions, and is optimally effective [3] given an admissible heuristic. However, A*'s disadvantages include its memory and time complexities [13]. The main drawback of A* is its memory requirements. A* stores in memory all the open nodes in order to guarantee admissibility, and all closed nodes in order to return the solution path once a goal is reached. The space complexity of A* is therefore equal to its time complexity which grows exponentially with the depth of the search. Therefore, A* cannot solve difficult problems since it usually exhausts all the available memory before reaching a goal. For the sliding-tile puzzles for example, A* can only solve the Eight Puzzle but not the Fifteen Puzzle on current machines. Like A*, KBFS also uses an open-list and will suffer from the same memory complexity.

This memory problem has been addressed in the last fifteen years, and several algorithms that generate nodes in a best-first fashion but use only a linear or a restricted amount of memory have been developed. Iterative-Deepening A* (IDA*)[13], introduced in 1985, is a sequence of depth-first search iterations. Each iteration expands all nodes having a total cost not exceeding a given cost threshold for that iteration. The threshold for the first iteration is the heuristic value of the initial state, and the threshold for each succeeding iteration is the lowest cost of all nodes generated in the previous iteration that exceeded the threshold for that iteration. IDA* generates new nodes in a best-first order only for *monotonic* cost functions, meaning that the cost of a child is greater than or equal to the cost of its parent. Recursive best-first search (RBFS) [14] is also a linear-space algorithm. It expands new nodes in a best-first order even when the cost function is *nonmonotonic*. The space complexity of both algorithms is linear in the maximum depth of the search, since they only need to store at most one branch of the tree and the siblings of nodes on that branch. Other attempts to search in a best-first order but with a limited amount of memory include MREC [25], MA* [1] and SMA* [24].

1.3 Weighted A*

The time complexity of A* or IDA* is proportional to the number of generated nodes, which is usually exponential in the solution cost. The time complexity can be very large when addressing large problems. For example, finding optimal solutions to some of the Twenty-Four Puzzle instances took several weeks to solve in [18] even though a heuristic function that is much more accurate than the Manhattan distance was used. One way to deal with this exponential complexity is to compromise on the quality of the solution and to settle for a suboptimal solution. With this compromise, a large reduction in the number of generated nodes can be obtained, at the cost of a small sacrifice in solution quality.

Weighted A* (WA*) [21] [7] is a generalization of A* which makes such a compromise. WA* uses $f(n) = w_g * g(n) + w_h * h(n)$ as its heuristic function. If we define w as the ratio w_h/w_g, then the cost function can be written in the equivalent form: $f(n) = g(n) + w * h(n)$. When $w = 1$ we get the A* algorithm, and when w is very large and the cost function is just $h(n)$ we get pure heuristic search. When $w > 1$, the overall cost function is inadmissible since it may overestimate the real distance to the goal and therefore the solution it provides may have greater cost than an optimal solution. It was shown in [14] that increasing w

WA*: Fifteen Puzzle			
Wg	Wh	Moves	Nodes
1	99	145.27	6,957
1	19	127.65	7,924
1	9	116.49	9,527
1	6	103.29	10,460
1	4	88.15	15,818
1	3	78.41	22,840
1	2	63.51	78,870
2	3	56.61	496,384
1	1	53.05	363,028,079

Table 1: WA*: results on the Fifteen Puzzle

(i.e. giving more weight to h and less to g) results in finding solutions faster in terms of the number of generated nodes, at the expense of an increased solution cost. It was proven in [2] that the solution cost returned by WA* cannot exceed the optimal solution cost by more than a factor of w. Empirical results [14] show that the solution cost of WA* is much smaller than this upper bound in practice. In fact, one can control the tradeoff between the cost of the path and the number of generated nodes by tuning w to any desired value. Table 1 illustrates this tradeoff of WA* for different solution lengths of the Fifteen Puzzle.[1] The values in the tables are the average over the standard set of 100 random problem instances that was first used in [13].

As shown in the table, a small sacrifice in solution quality yields a large improvement in running time. For example, obtaining a solution that is only 8% longer than the optimal solution yields an improvement of almost 3 orders of magnitude in the number of generated nodes.

1.4 Overview of the paper

In the following sections we introduce the K-best-first search algorithm, KBFS.

KBFS(k) is a generalization of BFS in that each expansion cycle of KBFS(k) expands the best k nodes from the open list. Their children are generated and added to the open list, and only then does a new node expansion cycle begin, expanding the k best nodes in the new open list.

We first describe the behavior of KBFS and claim that when there are large errors in an inadmissible heuristic function, KBFS outperforms BFS. Then, we show empirical results on incremental random trees, on the sliding-tile puzzle, and on the number-partitioning problem that confirm our claim.

Large problems, such as those larger than the Twenty-Four Puzzle, can only be solved suboptimally within a reasonable amount of time. Furthermore, while an optimal solution is ideal, in practice suboptimal solutions are usually sufficient for most purposes. We propose

[1]The last line of the table was calculated using IDA* because A* would exhaust the available memory. We would expect A* to generate fewer nodes than IDA*, since A* prunes duplicates nodes while IDA* does not. However, the number in the table gives a general impression of the number of generated nodes that A* would generate for optimally solving the Fifteen Puzzle.

KBFS as a simple but powerful algorithm for these purposes.

2 Attributes of KBFS

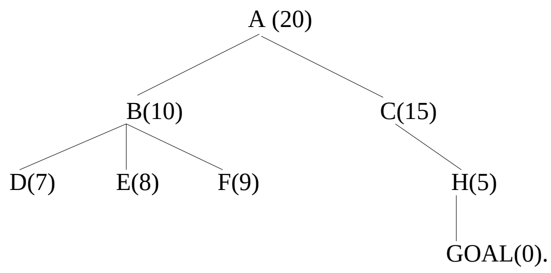

Figure 2: Example search tree for KBFS

KBFS(k) is a generalization of BFS in the sense that in each node expansion cycle, KBFS(k) expands the best k nodes from the open list. KBFS(1) is simply BFS. The number of nodes expanded in each iteration is the *width* or simply the k parameter. The difference between BFS and KBFS is that in BFS, after the children of the best node are added, a new expansion cycle begins and if one of these children is the best it is selected and expanded. In KBFS, the new children of a node are not examined until the rest of the previous K-best nodes are expanded and their children are added to the open list. KBFS(∞) is actually breadth-first search since at each expansion cycle all the nodes in the open list are expanded. All these nodes are at the same depth of the search tree. In that sense KBFS is a hybrid of best-first search and breadth-first search. BFS makes its decisions of which node to expand while relying completely on the cost value of the best node. As we will show below, if that value is mistaken, BFS can be lead astray. KBFS relaxes that dedication and tries to explore other possibilities as well. KBFS introduces diversity into the search. The constant time per node expansion for KBFS is the same as for BFS.

Consider the behavior of KBFS(2) versus BFS on the search tree of Figure 2, where the numbers represent the complete evaluation of each node. BFS expands node A generating nodes B and C. Then, it expands node B generating nodes D, E and F. It expands nodes D, E and F and only then goes back to expand node C generating node H and then the GOAL node. KBFS(2) however, after generating nodes B and C, expands them both generating nodes D, E, F and H. Then, nodes H and D are expanded and the GOAL node is reached. BFS explores the subtree under node B until all frontier nodes in that subtree have costs greater than node C. KBFS(2) looks at both subtrees simultaneously and reaches the goal faster in this case.

A cost value given to a node is based on a heuristic estimation of the real cost from that node to the goal, and therefore is not exact by nature. Thus, if a heuristic value of a node does not equal the actual distance to the goal we can say that there is an *error* in the heuristic. However, a good heuristic function is somewhat correlated with the real distance and usually gives a general idea about it and the errors in these cases are rather small. However, if there is a large gap between the heuristic value of a node and the exact cost, we can say that there is a *large error* in the heuristic estimation of the node.[2] Heuristic values of neighboring nodes are usually correlated with each other. A *region with large errors* is a neighborhood of nodes such that they all (or at least many of them) have large errors in their heuristic values. This scenario occurs when there is an attribute of the search space in that region that was not considered by the heuristic function. Entering a region with large errors might give a wrong impression that the a goal node is much closer to that region than what it really is.

When BFS chooses a best node for expansion it expects that the a goal node is a descendent of that node. However, when the heuristic value of the best node has a large error, this expectation may not be true. This might cause BFS to enter a region with large errors and thus search a direction that does not lead to a goal node. When such a scenario occurs we can say that BFS made a *poor decision* by choosing to expand a node in a wrong direction. A poor decision is choosing to expand a node where the expansion of another node would cause the search to reach the goal faster, i.e., with less computation effort.

When B and C in Figure 2 are at the front of the open list, BFS makes a poor decision and expands nodes under B while the goal is under C. In such cases, BFS may incur unnecessary overhead by expanding nodes in a wrong subtree. When that subtree is large and the values of the nodes in that subtree also contain large errors, then the overhead grows. On the other hand, when the goal node is indeed under the best node as suggested by the heuristic function, then BFS makes a correct decision in expanding it. In that case KBFS(k) incurs the overhead of expanding $k-1$ unnecessary nodes for that cycle, since expanding only the best node will suffice. If the value of node B were 22 for example, BFS would expand A,C,H and get to the GOAL. KBFS(2) would expand A,C,B,H and D and only then get to the GOAL. In this case, when BFS makes the correct decision, KBFS(2) expands two more nodes than BFS (B and D), one on each cycle. KBFS can be viewed as insurance against staying in a misleading subtree. The question is of course whether this insurance is worth the cost. The question of which algorithm causes more overhead arises: KBFS while exploring unnecessary nodes when correct decisions are made by BFS, or BFS when entering a wrong subtree after making poor decision. The main hypothesis of this paper is therefore:

> "As the number of large errors in the heuristic values of nodes in the search space increases, BFS makes more poor decisions and therefore KBFS with greater widths tends to outperform BFS".

One observation must be made here before continuing. KBFS will not be effective when the heuristic function is *admissible* and therefore the corresponding cost function is usually *monotonic*, meaning that the cost of a child is greater than or equal to that of their parent. To ensure obtaining optimal solutions, an admissible heuristic is required. To guarantee optimality, all nodes whose cost is less than the optimal solution cost must be expanded. Therefore, the order that they are expanded does not really matter. In the above example,

[2]For an admissible heuristic, the typical error will be a heuristic value that is much smaller than the real distance.

if the cost function had been monotonic, then generating the subtree of B would eventually reach a node with the same value as C, and then C would be expanded. With monotonic cost functions, BFS tends to correct itself when there are errors in the heuristic values. However, when the cost function is *nonmonotonic*, BFS may never reach C before the whole subtree of B is explored, and therefore KBFS is preferable here. A* for example, is normally used with an admissible cost function, and therefore we do not believe that KA* will systematically outperform A*. KBFS should be considered for cost functions that are not admissible, such as the weighted cost function of WA* [21] [14], where optimal solutions are not required, or are too expensive to calculate.

3 Related work

The idea that errors in the heuristic values can lead to poor behavior of ordinary search strategies has been examined by a number of researchers in the past few years. Limited Discrepancy Search (LDS)[9] was introduced in 1995. Given a heuristic preference order among successors of a node, a discrepancy is to not follow the heuristic preference at some node, but to examine some other node. LDS first looks at the path with no discrepancies at all, then all paths with one discrepancy, two discrepancies, etc. It was shown that LDS outperforms depth-first search (DFS) on some hard problems. This approach of LDS has been enhanced by Improved Limited Discrepancy Search [15] and Depth-Bounded Discrepancy Search [27].

Parallel Depth-First Search (PDFS) [23] is also related to KBFS. PDFS activates a number of DFS engines simultaneously. It was shown that in difficult problems where the heuristic function makes many mistakes at shallow levels, parallel DFS outperforms sequential DFS.

An interesting simultaneous DFS is Interleaved Depth-First Search [19]. It searches a path until a leaf is found, and if that leaf is not the goal, it does not backtrack to the father of the leaf, but rather starts another DFS path from the root node. The new DFS will branch differently then previous ones at the root node thus searching different parts of the tree more quickly. IDFS was shown to outperform DFS on randomized constraint satisfaction problems (CSP).

None of the above algorithms always follow the advice of the heuristic function. Sometimes they expand a node not recommended by the heuristic function. Nevertheless, they all try to improve on DFS and were tested on CSP trees where all goals are at the lowest level. Once reaching this level the algorithms can backtrack. Our work is novel in that we try to improve the behavior of BFS when there are large errors in the heuristic function.

The term 'K-best' is not new in the field of heuristic search and has been used to describe other algorithms. However none of these algorithms generalizes best-first search to its K-best version. For example, an algorithm similar to KBFS is the classic K-beam search. K-beam search is a best-first search that simultaneously expands the best K nodes from the open list. However, the size of the open list is limited to K. Only the best K nodes are stored on the open-list while other nodes are discarded. Even though this algorithm is reminiscent of our algorithm it is completely different. The new K nodes in K-beam search are chosen from a limited number of nodes. K-beam search never moves to another part of the tree, but rather only makes local decisions of where to continue. Therefore the decision of which K nodes to expand is very important in K-beam search, since it is unrecoverable. It is usually used for real-time applications or certain CSP problems. KBFS however, behaves in a best-first manner and chooses the K-best among all generated nodes. KBFS can always backtrack to another part of the tree when it realizes that it is proceeding in a poor direction.

K-beam search was used in CRESUS, an expert system for cash management [26]. They also added 'diversity' to the search by choosing to also expand nodes that seem very different from other nodes in the set of the K nodes that are chosen for expansion. Their motivation was that errors in the heuristic lead the search towards a 'local minimum' at the expense of getting to the 'global optimum' faster. Introducing diversity into the search helps by trying other directions as well. In that sense KBFS also adds diversity into the search, which might help to escape local minima.

Many attempts have been made to generalize best-first algorithms to their parallel versions. PIDA* [22] for example is one of these attempts that implements IDA* in Parallel. A closely related parallel best-first search to KBFS is PRA* [5]. In PRA*, each generated node is assigned by a hash function to one of the working processors and to a bucket of memory related to that processor. Each processor then expands the best node from its bucket, generates its children and broadcasts each of them to the processor they are assigned to by the hash function. Each processor keeps the children assigned to it in its bucket. If there are K processors, then some K promising nodes are always selected and expanded. However PRA* is different from KBFS in terms of which nodes it expands. In PRA* each processor locally chooses the best node from its bucket and expands it, while KBFS globally chooses the best K nodes among all the nodes in the open list. Also, PRA* was designed for a parallel architecture and will not be useful on a serial machine. PRA* is an admissible algorithm while KBFS is inadmissible. PRA* has a complicated retracting mechanism and incurs the overhead of communications between different processors, while KBFS is very simple to understand and implement.

4 Search trees with dead-ends

A special case of a large error in a heuristic is when a node has a low heuristic value implying that it is close to a goal, but in fact it is a root of a *Dead-end* subtree. A dead-end subtree is one that does not contain a goal node. Dead-end subtrees can occur in real-world problems. For example, in road navigation every peninsula or neighborhood surrounded by a fence is usually a dead-end subtree. *Pure heuristic search*, which is BFS with $f(n) = h(n)$, might visit every little town in the peninsula before trying to go around. KBFS will also explore nodes outside that subtree and therefore may reach the goal faster. A special case of a dead-end is a deadlock, which is a dead-end that cannot be recovered from. For example, the game of Sokoban[10] contains numerous deadlocks.

Consider the graph of Figure 3 where the value written next to a node represents its cost value. The two subtrees rooted at B and C are structurally symmetric, but node B is the root of a dead-end subtree. KBFS(1), i.e., BFS, might expand the nodes in the following order[3]: A, B, F, M, L, K, Q, E, D, C, H, N, Goal. It expands all the nodes of the left subtree

[3] The exact order depend of course on the tie breaking rule. However, the following order is a possibility.

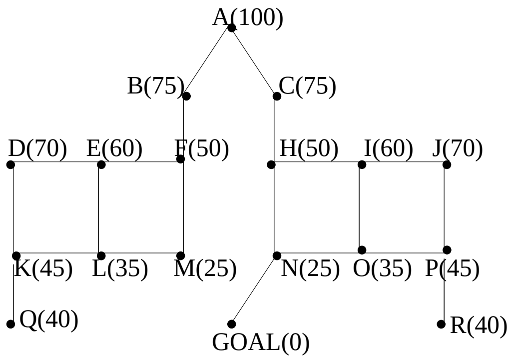

Figure 3: A graph with a dead-end: The whole subtree rooted at node B is a dead-end .

since their values are all smaller than the value of C, which is 75. Only then does it move to the subtree rooted at C. KBFS(2), however, expands the nodes in the following order: A, {B, C}, {F, H}, {M, N}, GOAL. Most of the nodes of the dead-end subtree of Figure 3 are not visited by KBFS(2).

4.1 Random trees with dead-ends

A common testbed for heuristic search algorithms are *incremental random trees*. They were used in [12], [20] and [17]. Such trees allow experimental control over the branching factor, search depth and the heuristic evaluation function. An incremental random tree is generated by assigning independent random values to the edges of the tree, and computing the value of a node as the sum of the edge costs from the root to that node. This produces a correlation between the heuristic value of a node and that of its descendants. The first tests that we conducted were on random trees that are very similar to incremental random trees, but the mechanism we used to generate them is a little different and will be described below.

In our trees, the value of a node represents the estimated heuristic distance to the goal. Thus we first assigned a value to the root of the tree and then for each node we assigned a random value that could not deviate from its parent's value by more than a constant value. This deviation is similar to the edge cost in the incremental trees. A node with a value of 0 or less is considered a goal node. In order to produce dead-end subtrees we installed a mechanism that introduces dead-end subtrees into the procedure that generates the tree. This mechanism randomly declares a node as a root of a dead-end subtree and then allows that subtree to have descendents only to a fixed depth. This depth is a variable parameter. All the heuristic values assigned to the nodes in such a subtree are errors since the real distance is ∞. Thus, this subtree is actually a region with large errors. We are aware of the fact that this is not a natural way to produce random trees and that we added a somewhat artificial dead-end subtree generator into the random trees. Our objective was to produce trees that simulate the phenomena of dead-end subtrees that occur in the real world, for example, in road navigation or in CSP problems.

For each of the random trees that we generated we used the following parameters:

1. The heuristic value of the root node was set to 2000.

DD	DFS	DFS+	bfs	kbf2	kbf3	kbf4	kbf5	kbf6	kbf7	kbf8	kbf9	kb10	kb11	kb12
nod	2.34	1.01	**1**	1.74	2.54	3.19	3.93	4.67	5.37	6.08	6.79	7.49	8.25	8.97
0	1.88	1.04	**1**	1.41	1.89	2.38	2.88	3.37	3.86	4.35	4.84	5.33	5.81	6.31
1	1.99	1.03	**1**	1.39	1.8	2.24	2.69	3.16	3.62	4.07	4.52	4.96	5.4	5.86
2	1.62	1.11	**1**	1.08	1.37	1.65	1.99	2.29	2.57	2.87	3.19	3.55	3.86	4.18
3	2.15	1.06	**1**	1.09	1.3	1.53	1.78	2.01	2.28	2.53	2.81	3.05	3.32	3.6
4	2.79	1.31	**1**	1.04	1.27	1.49	1.59	1.76	1.93	2.14	2.35	2.56	2.78	2.98
5	3.61	0.98	**1**	1.18	1.28	1.25	1.29	1.5	1.58	1.66	1.79	1.89	2.1	2.23
6	2.66	1.30	1	**0.92**	0.95	0.96	0.96	0.93	0.96	1.02	1.08	1.14	1.23	1.28
7	2.51	1.36	1	0.66	0.66	**0.56**	0.56	0.56	0.57	0.63	0.64	0.63	0.64	0.67
8	2.15	1.32	1	0.77	0.51	0.45	**0.39**	0.44	0.55	0.56	0.50	0.52	0.67	0.48
9	4.21	1.57	1	0.68	0.84	0.83	0.83	0.68	0.63	0.59	**0.54**	0.57	0.58	0.59
10	3.84	1.26	1	0.83	0.71	0.50	0.44	0.40	0.38	0.37	0.35	0.34	**0.32**	0.33
11	4.06	1.09	1	0.56	0.42	0.37	0.32	0.29	0.25	0.22	0.22	0.21	0.19	**0.18**
12	4.04	1.14	1	0.29	0.27	0.23	0.18	0.15	0.14	0.23	0.13	0.09	0.08	**0.07**

Table 2: KBFS: Random trees with variable depth of dead-ends

2. The maximum branching factor was set to 5. Each node has a random number of children uniformly distributed from 1 to 5.

3. The maximum deviation of a node's value from that of its parent was set to 50. In pure heuristic search, the cost values usually decrease as search gets closer to the goal. In order to simulate that we forced 80% of the nodes to have smaller values than their parents. This causes the cost function to be nonmonotonic.

4. The dead-end probability was set to 20%. Thus, each node not already inside a dead-end subtree has a probability of 20% to be declared the root of a dead-end subtree.

5. The maximum depth of a dead-end subtree is labeled DD (Dead-end Depth). Dead-end subtrees have depths uniformly distributed from 0 to DD. For example, a DD of 5 means that the subtree under each node that was selected to be the root of a dead-end subtree had a maximum depth of 5 or less. A DD of 0 means that the root itself is a dead-end and has no children at all. During the tests the DD parameter was varied from 0 to 12.

After the tree was generated, we searched for a goal node using different widths of KBFS. We counted the number of generated nodes for each algorithm. For comparison, we also searched with simple depth-first search, which chooses to expand an arbitrary child, and with depth-first search with node ordering, where children are chosen in increasing order of their heuristic values.

4.2 Test results

The variable parameter on our tests was DD increasing from 0 to 12. The row labeled with 'nod' is where there were no dead-ends at all. The results are shown in Table 2. Each row corresponds to a different value of the DD parameter. Each column specifies the results of different search methods: simple DFS, DFS with node ordering, BFS, and then KBFS(2) up to KBFS(12). The values represent the ratio of the number of nodes generated by each algorithm divided by the number of nodes generated by BFS averaged over 500 different

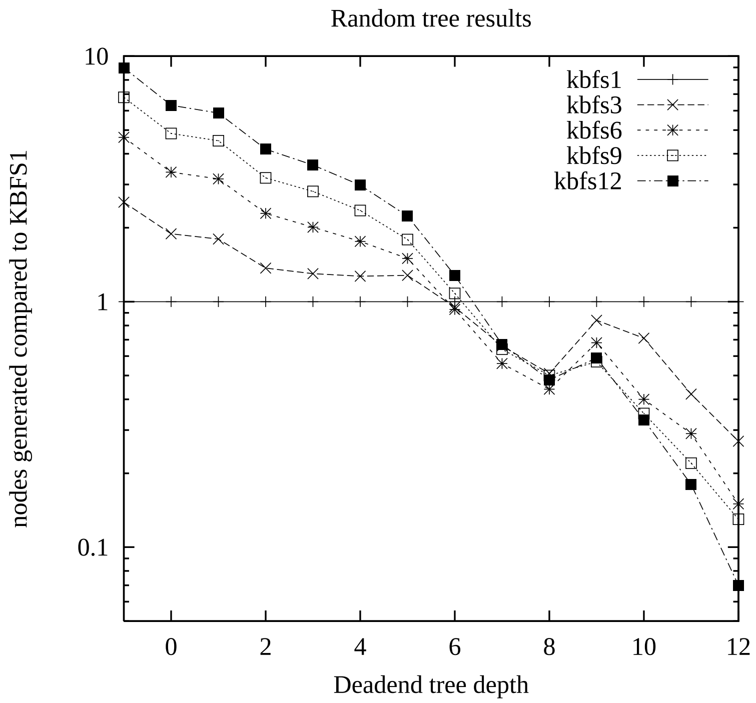

Figure 4: Results on the dead-end trees. Each curve is a different KBFS. The figure shows that when the depth of the deadend subtrees grows, KBFS with larger K tends to generate fewer nodes.

random trees. When that ratio is smaller than 1, then the tested algorithm outperforms BFS. The **bold** values in Table 2 represent the algorithm that performed best for each row (DD).

It is clear from Table 2 that as the dead-end subtrees become deeper, larger widths should be used for KBFS. In the first row where there were no dead-ends at all, BFS is the algorithm of choice. In this case the heuristics are usually accurate and thus larger widths for KBFS yield greater overhead. As long as the depth of the dead-end subtree, DD, does not exceed 5, BFS is the best option, because there are not enough nodes under a subtree with large errors to make it worth the overhead of expanding the other best nodes. From depth 6 and up, however, BFS is no longer the best. As the dead-end subtrees get deeper and deeper there are more nodes with large errors which make it more worthwhile to expand other best nodes, i.e. increase the width of KBFS. When DD is 12, KBFS(12) expands only 7% of the number of nodes expanded by BFS, representing a speedup factor of 15. Both Figure 4 and Figure 5 illustrate the data presented in Table 2. Figure 4 presents the overhead of KBFS versus the depth of the dead-end subtrees (DD). Each curve represents a different value of K. Figure 5 presents the overhead factor versus the width of KBFS. Each curve corresponds to a different DD.

Results of additional simulations with other values for the parameters of the tree were similar to the above results. The results from the random trees support our hypothesis that greater widths should be used where there are deeper deadends in the search space.

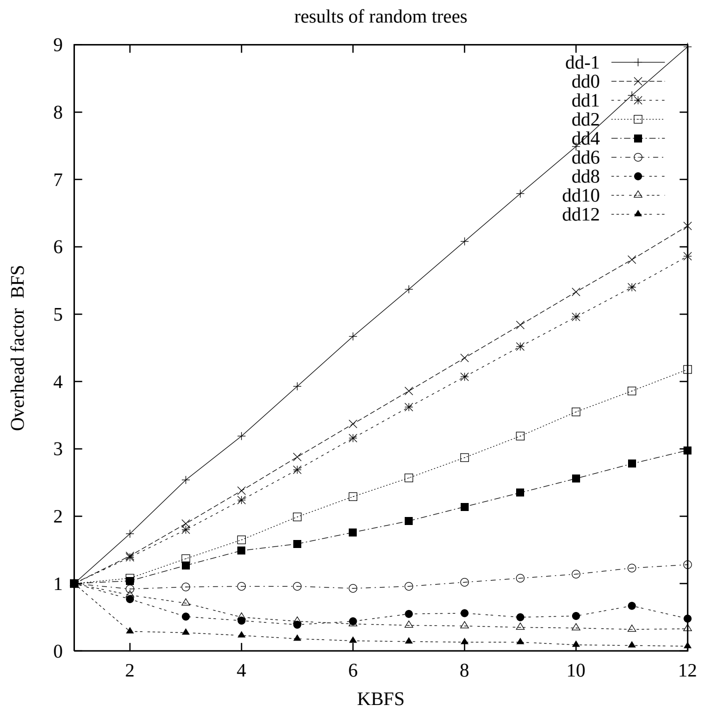

Figure 5: Results on the dead-end trees. Presents the overhead factor versus the width of KBFS. There is one curve for each DD. The figure shows that for large values of DD, larger KBFS are better choices.

5 KBFS results on sliding-tile puzzles

We also tested KBFS on sliding-tile puzzles with the Manhattan distance heuristic.

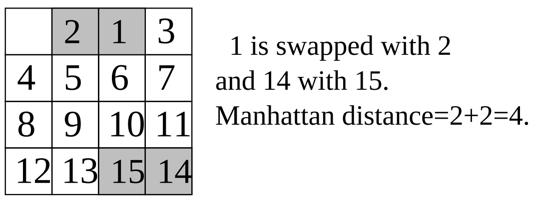

Figure 6: Example of a large error in the Manhattan distance heuristic for the Fifteen Puzzle

The Manhattan distance may have large errors for many states. For example, consider a state where all but two pairs of tiles are correctly placed but each of the tiles in these two pairs are swapped as in Figure 6. The heuristic value given to that state by the Manhattan distance function will be 4 while the actual number of moves required is 28. This large gap is because other tiles besides these two pairs must move as well. Nodes in this class behave like roots of dead-ends in the sense that the heuristic value given to them is very small compared to the correct distance to the goal. The heuristic values of many nodes in a subtree under such nodes contain large errors. Looking down into their subtrees will cause a lot of overhead and a goal node will not be reached as quickly as was suggested by the

W=	WA*		KWA*(10)		KWA*(50)		KWA*(100)		KWA*(200)		KWA*(2000)	
wh/wg	Path	Nodes	Path	Nodes	Path	Nodes	Path	Nodes	Path	Nodes	Path	Nodes
99/1	329.96	55853	285.62	59061	**224.7**	60934	199.76	94647	180.28	99349	130.68	550933
24/1	302.54	71500	257.28	87963	206.74	84322	**190.3**	82075	174.88	104097	130.68	550933
19/1	286.86	97697	242.56	103464	198.98	93488	183.78	92369	**168.0**	106272	130.74	551162
47/3	274.50	91208	237.48	128308	193.12	97641	**180.4**	91839	166.90	106157	130.74	551489
23/2	264.68	118795	226.66	143079	187.68	112678	175.74	114625	163.84	129115	130.56	555447
9/1	248.86	146251	215.96	168899	181.06	122100	170.34	111062	161.26	145327	130.50	559140
8/1	243.18	131928	209.38	218125	177.54	143871	169.78	132574	159.72	150326	130.56	558858
7/1	235.42	154343	204.46	235661	173.82	166265	166.90	153426	157.38	167788	130.58	562409
87/13	231.56	189788	200.84	276119	172.16	181486	165.96	182249	156.32	169520	130.52	564705
6/1	**226.1**	227516	195.74	368922	169.54	171483	163.38	187080	154.36	208463	130.08	579769
17/3	221.18	321137	192.74	348828	168.22	170398	161.72	196442	153.66	221126	130.01	591060
5/1	211.60	297459	184.40	495825	163.70	236703	158.88	282682	151.90	271918	129.62	631814
41/9	200.26	340404	179.02	457011	159.52	249794	156.00	337065	149.36	291430	129.44	647989
4/1	**191.9**	468727	172.64	527791	155.54	369183	151.10	403235	146.32	444432	128.46	715038
39/11	**180.5**	652085	164.94	754533	150.20	520438	147.18	489938	142.50	637711	128.44	884604
19/6	**169.9**	781549	157.58	867552	144.29	764153	140.76	776490	139.38	784237	127.40	997267

Table 3: KBFS: results on the Twenty-Four Puzzle

heuristic value. KBFS might overcome this overhead because the value of another best node might be more accurate and the algorithm may bypass the problem.

We tested KBFS both on the 4×4 Fifteen Puzzle and on the 5×5 Twenty-Four Puzzle with the WA* heuristic function, $f(n) = g(n) + w * h(n)$. For $h(n)$ we used the Manhattan distance heuristic. We ran the resulting algorithm, called KWA*, with different values for K and w. For w we used the same values that were used in [14]. Each KWA* algorithm was run over the set of 100 instances of the Fifteen Puzzle that were first used in [13]. For the Twenty-Four Puzzle we generated 100 solvable instances. Both the solution length and the number of nodes generated by each of these runs were recorded and then averaged over these 100 instances.

5.1 Results on the Twenty-Four Puzzle

On the Twenty-Four Puzzle we tested 30 different values of w on 100 randomly-generated solvable problem instances producing 3000 different search trees. The tests were performed on a 233MHz pentium with the ability to store up to 13 million states in its memory. Table 3 shows the results of KWA* on the Twenty-Four Puzzle. Each number represents the average of the 100 instances for different values of K and w. When using simple WA*, which is a best-first search, smaller values of w produce shorter solutions at the expense of more generated nodes as shown in the first two data columns. The results from Table 3 suggest a better method for getting shorter solutions, namely increasing the width of KWA*. The table shows that KWA* can produce the same solution length as WA* but with an improvement of up to a factor of seven in the number of generated nodes. The bold values represent examples of pairs of data points with nearly equal path lengths but with smaller numbers of generated nodes. For example, WA* finds a solution with a length of 226.1 moves while generating 227,516 nodes, while KWA*(50) finds a path of length 224.7 with only 60,934 generated nodes. For a length of 192 moves, WA* generates 468,727 nodes while KWA*(100) needs only 82,075 nodes. For a length of 180 moves WA* generates 652,085 nodes while KWA*(100) generates only 91,839 nodes. For a length of 170 moves, WA* needs 781,549 nodes while KWA* generates only 106,272 nodes.

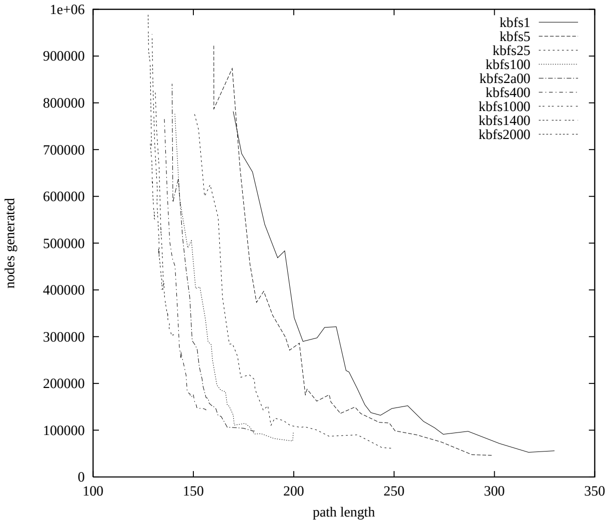

Figure 7: Path length versus generated nodes for Twenty-Four Puzzle

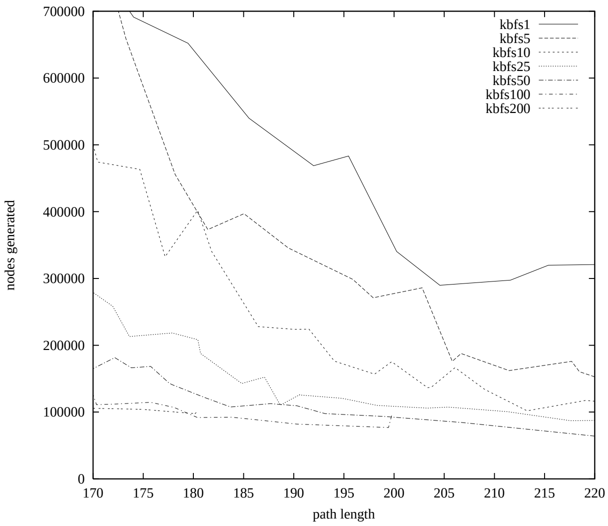

Figure 8: Path length versus generated nodes for the Twenty-Four Puzzle. Larger scale

Figure 7 and Figure 8 present a comparison of different KWA* algorithms by plotting node generations as a function of solution length. Each curve corresponds to a different value of K. The data in Figure 8 is the same as that in Figure 7, but on a larger scale. The figures show that WA* is indeed outperformed by all other values of KWA*.

As stated earlier we had 3000 different search trees (100 initial states with 30 different values for w). In each such tree we can compare the behavior of the tested algorithms. Table 4 presents the percentage of instances of the 3000 in which each algorithm beats WA*. The following data is included in the columns of the table:

KWA*	Both	O len	O gen	None	Len	Gen
1	0.00	0.00	0.00	0.00	0.00	0.00
2	31.41	25.22	17.89	25.48	56.63	49.30
3	36.96	27.93	14.41	20.70	64.89	51.37
5	40.81	33.70	9.04	16.44	74.52	49.85
10	42.96	40.78	4.04	12.22	83.74	47.00
20	48.11	45.78	1.41	4.70	93.89	49.52
30	49.93	47.52	0.11	2.44	97.44	50.04
40	48.93	49.07	0.19	1.81	98.00	49.11
50	49.56	49.44	0.19	0.81	99.00	49.74
100	45.52	53.81	0.04	0.63	99.33	45.56
200	37.74	62.22	0.00	0.04	99.96	37.74
400	30.89	69.11	0.00	0.00	100.00	30.89
700	22.67	77.33	0.00	0.00	100.00	22.67
1000	18.22	81.78	0.00	0.00	100.00	18.22
2000	11.30	88.70	0.00	0.00	100.00	11.30

Table 4: KBFS: wining and losing on the Twenty-Four Puzzle

- **Both:** represents the percentage of instances out of 3000 that an algorithm generates both fewer nodes and a shorter solution compared to WA*.
- **Only len:** counts the cases in which an algorithm produces a shorter solution length but generates more nodes.
- **Only Gen:** Fewer nodes but longer solution.
- **None:** More nodes and longer solution.
- **Len:** This column counts the percentage of search trees for which the algorithm produced a shorter solution compared to WA* without taking into consideration how many nodes were generated. This column is the sum of the first two columns.
- **Generated:** Counts the percentage that each algorithm generated fewer nodes regardless of the solution length. This number is the sum of the first and third columns.

When comparing two algorithms we define a *win* as a case that one algorithm generated fewer nodes and a shorter solution and a *loss* as a case were the algorithm generates more nodes and a longer solution. Thus, the Both and None columns of Table 4 represent winning and losing, respectively, by these definitions. If we compare the Both column to the None column we see that KWA*(2) is the worst KWA* on the Twenty-Four Puzzle. It beats WA* in 31.41% of the cases while it loses in 25.48%. Still, it wins more than it loses. KWA*(30) for example wins 49.93% and loses only 2.44%. For a width of 400 and up, KWA* always produces shorter solutions, often with fewer numbers of generated nodes as well.

5.2 KBFS results on the Fifteen puzzle

On the Fifteen Puzzle we tested each KBFS on 44 different ratios of w. The values for w were taken from [14]. Each pair of K and w (KWA*) was tested on the 100 random instances. Here, the improvement factor of KWA* over WA* was 2-3. For example, when w was set to 1/3 WA* finds a solution of length 78.41 moves at a cost of 22,840 node generations, while KWA*(50) finds a path of length 77.41 moves with only 9,987 generated nodes with $w = 1/9$. For a path with a length of 68 moves, WA* generates 45,736 nodes with $w = 3/7$ while KWA*(100) generates only 18,896 nodes with $w = 1/4$. The data on winning and losing in the Fifteen Puzzle was not significantly different from that of the Twenty-Four Puzzle. We do not present the data of the Fifteen Puzzle in more detail since the Fifteen Puzzle can be solved optimally very fast by algorithms such as IDA*, so near-optimal solutions are no longer interesting for this version of the puzzle.

The results on the Twenty-Four Puzzle are significantly better than on the Fifteen Puzzle. The KBFS technique is not only more powerful on larger problems, but is also more valuable there since finding an optimal solution for large problems can be very expensive. For example, some instances of the Twenty-Four Puzzle take weeks to solve optimally, even with a more sophisticated heuristic than Manhattan distance. In such problems, settling for a near-optimal solution is the only practical way alternative within reasonable time constraints.

We also tried to solve the 6x6 35 puzzle with KWA*. Nevertheless, even with pure heuristic search, i.e. $f(n) = h(n)$, none of the algorithms could solve an entire set of 100 random problem instances with the available memory, which could save up to nine million nodes. Only 85% of the instances could be solved and the partial results on these 'easy' instances show the same tendency, namely KWA* outperforms WA*.

6 KBFS on number partitioning

Neither the sliding-tile puzzles nor the random trees are real-world problems. Furthermore, since KBFS does not return optimal solutions, it might be compared with other algorithms which also do not guarantee optimal solutions. For these reasons, we have implemented KBFS on the number partitioning problem and compared it to hill-climbing which also does not guarantee optimal solutions.

Given a set of N numbers, the two-way number-partitioning problem is to divide them into two disjoint subsets, such that the sum of the numbers in each subset are as nearly equal as possible. This problem has applications in the real world. For example, given two machines, a set of jobs and the time required to solve each job on either machine, the task is to assign the jobs to the machines such that all jobs are completed in the shortest elapsed time. This problem is NP-complete [6].

There are many algorithms that were specially designed to solve this problem. One of these algorithms is the set differencing algorithm also known as the Karmarkar-Karp or the KK[11] heuristic. The KK algorithm first sorts the numbers. Then, it removes the largest two numbers, computes their difference and treats the difference as another number, inserting it into the sorted list of numbers. This is equivalent to separating the two largest numbers in different subsets, since putting one of these numbers in one subset and the other one in the other subset is equivalent to putting their difference in one of the subsets. KK stops when there are no numbers left in the sorted list. KK runs in $O(n \log n)$ time, but doesn't return optimal solutions.

A fast optimal algorithm was proposed in [16]. It was called CKK which stands for Complete Karmarkar-Karp algorithm. The basic KK takes the largest two numbers and places their difference in the sorted list indicating that these numbers are being placed in different subsets. The only other option is to place their sum in the sorted list to indicate that both numbers are to be placed in the same subset. CKK explores both these options and thus finds an optimal solution to the problem. It outperforms all known algorithm for large number partitioning problems. Our goal in implementing KBFS on this problem is not to try to compete with algorithms that are specially designed for this problem, such as CKK. Rather, our intention is to check whether increasing K improves search in this domain.

6.1 The search tree for number partitioning

There are 2^N possible ways to partition N numbers into two disjoint sets. A state in our search space is a complete partition of the numbers into two disjoint sets. The cost of a state is simply the difference between the sum of the numbers in each of the subsets.

We specify two types of neighbors of a state:

1. A single number may be moved to the other subset while all the other numbers remain stationary. Thus, the number of neighbors of this type for every state is N.

2. Sometimes, moving a number from subset A to subset B will produce a large increase in the cost of a state. However, moving another number from B to A will cancel this increase and might even decrease the cost. In other words, sometimes swapping two numbers might decrease the cost. Thus, whenever moving a number to the other subset increases the total cost we look at all the possibilities to swap this number with numbers from the other subset. We therefore also consider all these swapping moves as generating neighbors of the original state. Note that we consider a pair swap as a neighbor only if moving either of the single numbers alone increases the cost. Thus only a subset of all the possible pair swaps is considered.

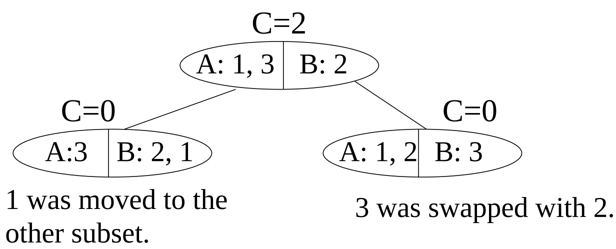

Figure 9: Number partitioning: the different types of neighbors.

Figure 9 illustrates both types of neighbors. There are three numbers to partition with values of 1, 2 and 3. In the root node, subset A contains 1 and 3 while subset B contains 2. Moving 1 to subset B is a neighbor of type 1, and the cost decreases from 2 to 0. This scenario is presented in the left child. Moving 3 to subset B increases the cost from 2 to 5. Thus, any possible way to swap 3 with members of subset B is also considered a neighbor. In particular, swapping 3 with 2 will decrease the cost from 2 to 0 as illustrated in the right

Algorithm	20	25	30	35	40	45	50	55	60
hill-climbing	62883	7768	4919	6837	5389	5543	5466	5405	5480
BFS	62883	6175	6192	7605	7794	10007	7638	9118	10664
KBFS3	62883	6175	6192	7805	7966	9851	7551	7793	7630
KBFS10	62883	6175	6189	7859	7118	6740	7767	5879	6290
KBFS30	62883	6175	6093	6962	5367	5827	5763	6252	5834
KBFS100	62883	6124	6206	4049	5795	5373	4665	4621	5359
KBFS300	62883	5406	4252	3924	4530	4737	4715	4598	4899
KBFS1000	62883	4336	3723	3547	4333	4400	3514	3646	3987

Table 5: KBFS: comparison with hill-climbing on number partitioning. The values are the average cost of a solution after generating 6,000,000 nodes.

child. When many neighbors of type 1 increase the cost, then the branching factor of our search space is very large and can contain $O(N^2)$ different neighbors[4].

KBFS is a very simple and general algorithm for traversing a search tree. Thus, a candidate algorithm for comparison should also be simple and general. We have chosen to implement hill climbing as it is the simplest local search algorithm and can be implemented for almost every domain. Hill climbing requires only O(N) memory where N is the number of numbers to be partitioned, while KBFS requires exponential space. Hill climbing was implemented with random restart as follows. We first generate a random state. Then, we generate all its neighbors and choose one with the lowest cost. If this cost is better than the previous cost, we discard all other neighbors and replace the current state with this new state. This process is continued until a local minimum is reached. Then, we generate another random state and perform this search again until another local minimum is reached. This hill-climbing procedure is repeated until some external condition is reached and then returns the best solution found. In our experiments the terminating condition was a bound of six million nodes generated. Once this many nodes were generated, the search stopped. The reason that we picked this bound is because KBFS stores all generated nodes in an open list simultaneously and this is the number of nodes that we could store in our memory.

After hill-climbing returns its best solution, we move to KBFS. We took the same initial random partitionings that were generated by hill-climbing and inserted them into the open list. Then, we activated KBFS with different values for K on this open list. In this way both hill-climbing and KBFS start the search from the same set of nodes. Hill-climbing performs local search for each such node and tries to reach a local optimum in its neighborhood. KBFS searches all these nodes simultaneously, comparing the cost of all the nodes and choosing the best K nodes to be expanded. KBFS also stops after 6,000,000 nodes were generated and returns the best solution that it found during the search.

6.2 Empirical results

Table 5 presents the results of KBFS on number partitioning. Each row corresponds to a different algorithm. The columns correspond to the size of the set of numbers that were partitioned which was varied from 20 to 60. The numbers to be partitioned were randomly generated from a uniform distribution from 0 to 10 billion, i.e. they contain up to 10 decimal

[4]The number of different pairs.

digits. The values in the table are the best solutions found by the algorithms after generating 6,000,000 nodes. The values are all averages of 100 instances of the same size.

The first column corresponds to a set of size 20. Since there are only $2^{20} = 1,048,576$ different ways to partition such a set then the whole search space is spanned by all versions of KBFS and an optimal solution is obtained. With larger sets only suboptimal solutions were obtained. While the difference between the solutions returned by the different algorithms is not very large, we can see here a tendency that increasing K, especially to large values such as 1000, finds better solutions. For example, for problems of size 35, KBFS(1000) returned a solution which is less than half the size of the solution obtained by both hill-climbing and BFS. The advantage of hill-climbing over KBFS is the fact that it is easier to implement. Also, KBFS uses an open list and incurs the overhead of both time and memory needs for handling this list. If we only compare BFS to KBFS, our results show that KBFS outperforms simple BFS.

7 Conclusions and future work

We have shown a simple method to increase the effectiveness of best-first search by generalizing it to KBFS. We have implemented KBFS on various domains and the results show that expanding several nodes at a time is more effective than expanding just the best node. The reason is because errors in the cost function handicap simple best-first search more than KBFS, since KBFS adds diversity to the search.

KBFS is a simple and general formulation for expanding nodes from an open list. Many problems can be solved with algorithms that were specially developed for these particular problems. KBFS, as a general algorithm, may not compete with these hand-tooled algorithms. However, if one chooses to use a simple and general algorithm such as best-first search, this paper suggests that one might also consider KBFS because it is as simple and general as best-first search, but may outperform it with no additional cost of machine resources nor implementation complexity.

KBFS might be further explored in the following directions:

1. Determine under what circumstances KBFS outperforms BFS and what is the optimal value of K.

2. A linear or memory-bounded space version of KBFS may be introduced in order for KBFS to be widely used. RBFS [14] and SMA* [24] might be good starting points.

3. A parallel version of KBFS might be examined. In this form of KBFS, K processors expand the K best nodes simultaneously.

References

[1] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(2):197–221, 1989.

[2] H. W. Davis, A. Bramanti-Gregor, and J. Wang. The advantages of using depth and breadth components in heuristic search. *Methodologies for Intelligent systems 3*, pages 19–28, 1989.

[3] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, 32(3):505–536, 1985.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] M. Evett, J. Hendler, A. Mahanti, and D.S. Nau. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2):92–103, 1995.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA, 1979.

[7] J. Gasching. *Performance measurement and analysis of certain search algorithms*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh Pa, 1979.

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

[9] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proc of IJCAI-95*, pages 607–613, Montreal Canada, 1995.

[10] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *Proc of IJCAI-99*, pages 570–575, Stockholm, Sweden, 1999.

[11] N. Karmarkar and R. M. Karp. The differencing method of set partitioning. *Technical Report, UCB/CSD 82/113*, 1982.

[12] R. Karp and J. Pearl. Searching for an optimal path in a tree with random costs. *Artificial Intelligence*, 21(1-2):99–116, 1983.

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

[14] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41 –78, 1993.

[15] R. E. Korf. Improved limited discrepancy search. In *Proc of AAAI-96*, pages 286–291, Portland Oregon, 1996.

[16] R. E. Korf. A complete anytime algorithm for number partitioning,. *Aritificial Intelligence*, 106(2):181–203, 1998.

[17] R. E. Korf and D. M. Chickering. Best-first minimax search. *Artificial intelligence*, 84(1-2):299–337, 1996.

[18] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134:9–22, 2002.

[19] Pedro Meseguer. Interleaved depth-first search. *Proc of IJCAI-97*, pages 1382–1387, 1997.

[20] D.S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial intelligence*, 21(1-2):221–144, 1983.

[21] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[22] C. Powley and R. E. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, 1991.

[23] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on parallel and distributed systems*, 4:427–437, 1993.

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

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

[26] P. Shell, J. A. H. Rubio, and G. Q. Barro. Improving search through diversity. In *Proc of AAAI-94*, pages 1323–1328, Seattle Wa., 1994.

[27] Toby Walsh. Depth-bounded discrepancy search. *Proc of IJCAI-97*, pages 1388–1393, 1997.