Single-Frontier Bidirectional Search

Ariel Felner
Information Systems Engineering
Deutsche Telekom Labs
Ben-Gurion University
Be'er-Sheva, Israel 85104
felner@bgu.ac.il

Carsten Moldenhauer
Computer Science Department
University of Berlin
Berlin, Germany
carsten.moldenhauer@googlemail.com

Nathan Sturtevant
Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{nathanst, jonathan}@cs.ualberta.ca

Jonathan Schaeffer

Abstract

On the surface, bidirectional search (BDS) is an attractive idea with the potential for significant asymptotic reductions in search effort. However, the results in practice often fall far short of expectations. We introduce a new bidirectional search algorithm, *Single-Frontier Bidirectional Search* (SFBDS). Unlike traditional BDS which keeps two frontiers, SFBDS uses a single frontier. Each node in the tree can be seen as an independent task of finding the shortest path between the current start and current goal. At a particular node we can decide to search from start to goal or from goal to start, choosing the direction with the highest potential for minimizing the total work done. Theoretical results give insights as to when this approach will work and experimental data validates the algorithm for a broad range of domains.

Introduction

Most start-to-goal search algorithms are unidirectional, i.e., they search from a start state towards a goal state. Bidirectional search (BDS) is a general framework where the search is performed simultaneously from the start and from the goal until the two search frontiers meet. BDS has proved to work very well in domains that can fit into memory and have no heuristic guidance. However, when heuristic guidance exists it has been shown that, in theory, traditional BDS has little potential to outperform unidirectional search (Kaindl & Kainz 1997). The reason is the *meet in the middle problem* of guaranteeing the optimality of the solution after the search frontiers meet. A number of non-traditional BDS algorithms have been proposed (Kaindl & Kainz 1997) but the implementation of these algorithms is usually difficult and the gains have not been impressive. Therefore, in practice, BDS is rarely used when heuristics are available.

Problems that require more memory than is available are usually solved with depth-first search (DFS) algorithms such as IDA* (Korf 1985). The classical idea of searching from both directions has not been broadly considered for DFS algorithms. The notion of meeting frontiers does not apply here since the frontiers are not kept in memory. We show how the meeting of the frontiers can be achieved even when DFS algorithms are considered.

We introduce a new type of bidirectional search called *Single-Frontier Bidirectional Search* (SFBDS). A node in a search tree using SFBDS consists of a pair of states, s and g, and corresponds to the task of finding the shortest path between them. This task is recursively decomposed by expanding either s or g and generating new tasks between (1) the neighbors of s and g, or (2) the neighbors of g and s. At every node a *jumping policy* decides which of the two states to expand next, i.e., the search can proceed forward or backward. Given a fixed jumping policy, a tree is induced which can be searched using any admissible search algorithm.

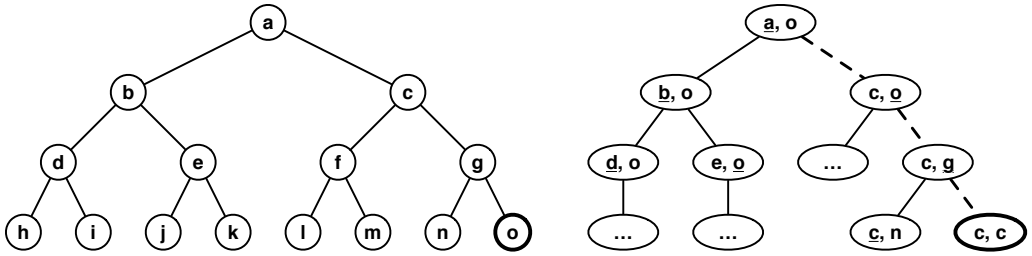
We first introduce SFBDS and show that it may reduce the size of the search tree by leveraging irregularities in the branching factor of a search space. These irregularities may naturally occur in a domain or appear as a result of heuristic pruning of the search space. Next, we show that SFBDS is a generalization of the *dual search* concept (Zahavi *et al.* 2008), a complicated algorithm which is limited to combinatorial puzzles with specific properties. Finally, we provide experimental results for depth-first and best-first implementations of SFBDS confirming our analysis.

Single-frontier bidirectional search

In this paper we use the term *node* and use capital letters (e.g. N) to indicate nodes of the search tree, while the term *state* and small letters (e.g., s) are used to indicate states (or vertices) of the input graph. We assume that the input graph is undirected and that the task of the search is to find a *shortest* path from the start state to the goal state. Our ideas can be generalized to work without these assumptions.

Unidirectional search

Assume the task is to find a path between s and g on a graph. Regular search algorithms formalize a search tree such that each node of the tree includes one state of the graph. The root node R includes the start state s. Assume that node N corresponds to state x. The task at N is to find a (shortest) path between x and g. When a heuristic is applied, it estimates the length of the path from x to g (h-cost) and adds this to the accumulated path from s to x, i.e., all edges from R to node N (g-cost). When the goal is reached via an optimal path, we backtrack and the states of the path are passed up the tree to construct the solution path.



(a) Unidirectional tree (b) Task tree

Figure 1: Example for unidirectional and SFBDS trees.

SFBDS: Formal Definition

The main observation underlying this paper is the following:

> Finding the shortest path between states x and g can be solved via recursive decomposition. The method used for the decomposition (direction of the search) does not matter as long as an optimal path is returned.

In SFBDS each node is defined as a pair of states x and y denoted by $N(x, y)$. The task of such a node is to find a shortest path between x and y. In other words, the task is to *close the gap* between x and y. This can be done by treating x as the start and y as the goal, searching from x to y. An alternative is to *reverse* the direction of the search by treating y as the start and x as the goal, searching from y to x. For example, if at $N(x, y)$ both x and y have two neighbors, then the children of N of the two alternatives are:

(a) regular direction (expand x): (x_1, y) and (x_2, y); or

(b) reverse direction (expand y): (x, y_1) and (x, y_2).

Each node N should be expanded according to one of these alternatives. The search terminates when a *goal node* is reached ($N(x, y)$ where $x = y$). The choice of search direction in N is reflected by N's children only, but no other node in the search is influenced by this choice of direction. Solutions or cost estimates from node N are naturally passed up to the parent of N, regardless of the direction used for N.

We use a *jumping policy* to choose which direction to continue the search at each node. Define the *task search tree* (*task tree* in short) for a given jumping policy as the tree obtained by using $R(s, g)$ as the root of the tree. Any admissible algorithm can be used to search for a shortest path from R to any goal node in the task search tree.

Examples

Unidirectional search and SFBDS are illustrated using the graphs in Figure 1. The objective is to find a shortest path from the start state, a, to the goal state, o. Consider a unidirectional search (Figure 1a). In this tree, every node implicitly solves the task of getting from the current node to o, and the search will proceed across the tree until o is found.

Now, consider searching the same tree with SFBDS (Figure 1b). Nodes are labeled with the shortest-path task that should be solved below them. The state which is chosen for expansion by the jumping policy is marked with an underscore. For example, at the root, the task is $(\underline{a}, o)$ resulting in two children, (b, o) and (c, o). At node $(c, \underline{o})$, however, the jumping policy chooses o for expansion. This generates nodes for all neighbors of o, leading to (c, g) in our example. Finally, at $(c, \underline{g})$, state g is chosen for expansion, generating a goal node (c, c).

Edges in a task tree are of two types. The first type are edges from a node (x, y) to a node (w, y) which corresponds to an edge (x, w) in the graph (expanding x). The second type are edges from a node (x, y) to a node (x, z) which corresponds to an edge (y, z) in the graph (expanding y). For example, the path in the task tree in Figure 1b indicated by dotted arrows corresponds to edges (a, c), (o, g) and (g, c). Constructing the solution path is straightforward. When backtracking up the search tree from a goal node, edges that correspond to forward expansions are appended to the front of the path while edges that correspond to backwards expansions are appended to the end of the path. Thus, the path of (a, c, g, o) is constructed from this branch.

Analysis

Given a specific jumping policy the task tree is determined. Every shortest path in the task tree encodes a shortest path in the graph and vice versa. Therefore, using any admissible search algorithm on the task tree will return an admissible solution for the original graph. No gains can be provided by using any other search algorithm besides A* (or any of its variants) because it is guaranteed to find the optimal path and the nodes it expands are mandatory. This applies to any type of search tree and to the task tree as well.

The main aim of SFBDS is to minimize search effort by choosing an appropriate jumping policy. Regular unidirectional search uses the policy *never jump*. Similarly, a unidirectional search from the goal to the start employs the policy *jump only at the root*. The idea is to improve upon these jumping policies. Unfortunately, the space of jumping policies is exponential in the number of nodes expanded and it is out of the scope to determine an optimal jumping policy at runtime. However, heuristic approaches can be used.

We distinguish four types of domains (shown in Figure 2) to consider if jumping at node (a, g) can be advantageous.

Case 1: Consider domains with a uniform branching factor b. A search without heuristic guidance on the unidirectional search tree will expand O(b^d) nodes when the solution is of length d. Consider the task trees generated by different jumping policies. Since the branching factor is uniform all the task trees have the same structure. Hence, searching a task tree of a different jumping policy will also require O(b^d) node expansions. In Figure 2a consider the policy where the search does not change direction until it reaches node (a, g)

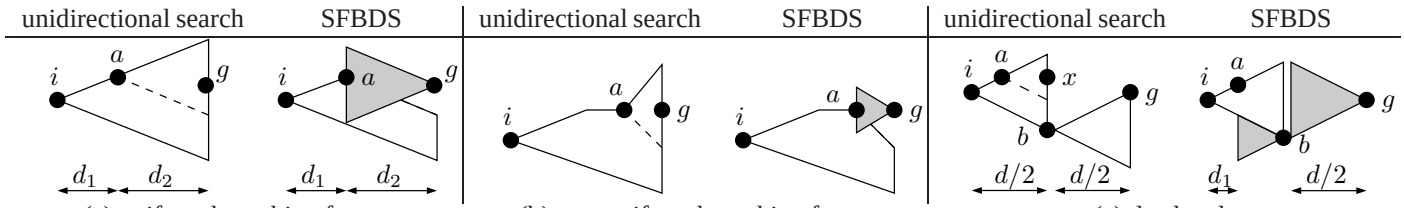


Figure 2: Case analysis when reversing the search direction can be advantageous.

at depth d_1. A unidirectional search would continue with a search to depth $d_2 = d - d_1$, expanding O(b^{d_2}) nodes below node (a, g). Now, assume the direction of search is reversed at (a, g). This search continues from g to depth d_2, also expanding O(b^{d_2}) nodes below node (a, g). Therefore, any jumping policy, including no jumping, is optimal.

Case 2: Consider domains with a non-uniform branching factor and without dead ends (Figure 2b). A non-uniform branching factor can be a domain characteristic or the result of heuristic pruning. If one area of a state space has better heuristic values than another, the effective branching factor (number of nodes expanded below a given node) in that area may be smaller. Hence, there is the potential to reduce the search effort by choosing a good jumping policy.

Case 3: Consider domains with dead ends in the search tree (Figure 2c). Consider a unidirectional search from i to g. At depth $d/2$ only b is connected to the goal and all other nodes (e.g., x) are dead ends. In this search we have two trees of depth $d/2$ and the total effort is O($2 \times b^{d/2}$) =O($b^{d/2}$). SFBDS with a non-optimal jumping policy does not recognize dead ends until at least one of the two states of a node is a dead end. When the search direction reverses below (a, g) it will search to an additional depth of $d - d_1$ before being pruned. Hence, the search tree is of size O($b^{d/2} + b^{d/2-d_1}$). Using the simplistic assumption that the search always reverses at depth d_1, the total effort will be O($b^{d_1}(b^{d/2} + b^{d/2-d_1})$) $\gg$ O($b^{d/2}$) before a solution is found. In these domains it is crucial to jump in the right places or not to jump at all to avoid a quadratic increase in the search effort. Good heuristic guidance can be used to prune the unsuccessful searches earlier than at depth $d - d_1$.

Case 4: On graphs with many cycles best-first search algorithms are quite effective due to duplicate detection. If there are V states in the graph, a best-first version of SFBDS can have up to V^2 unique tasks, so there is a potential asymptotic increase in the size of the state space.

Relationship to Dual Search

The concept of *duality* and an algorithm called *dual search* (DS) (Felner *et al.* 2005; Zahavi *et al.* 2008) was introduced in the context of permutation state spaces such as Rubik's Cube. Assume that when applying the operator sequence O to s we arrive at g. The *dual state* of s, denoted as s^d, is defined to be the state which is reached by applying O to g. The distances from s and s^d to g are equal. Therefore, any admissible heuristic lookup for s^d is also admissible for s. Given a heuristic h, $h(s, g)$ is called the *regular lookup* and $h(s^d, g)$ is called the *dual lookup*. If they differ their maximum yields a more powerful heuristic.

DS exploits the heuristic differences of the two lookups. At each state, it either continues to search regularly or it "jumps" to the dual state and continues the search from there. For pattern databases (PDBs), a good jumping policy is to "jump" if $h(s^d) > h(s)$. This was called the *jump if larger* (JIL) policy. Dual IDA* (DIDA*) was shown to significantly outperform IDA* in many combinatorial puzzles.

DS has two main limitations. First, it only works in domains (e.g., combinatorial puzzles) which have the special property that each operator corresponds to a *location-based permutation*. Similarly, it assumes that the same type of operators are applicable to all states. Second, the concept of the dual state and the DS algorithm are technically complicated and hard to understand. In addition, DS was only implemented on top of IDA* and the question as to whether it is applicable to A* remains open.

SFBDS generalizes DS to all possible state spaces and is simpler to understand. Many DS concepts, such as the distinction between "simple" and "general" duality, disappear when viewed as a special case of SFBDS. This paper also shows that SFBDS is suitable for search with A*.

To understand why DS and SFBDS are equivalent let O be the *location-based* permutation that transfers s into g. That is, assume location $x \in s$ occupies object a. O specifies the destination location $y \in g$ for object a. Now, $O(s) = g$ and $O^{-1}(g) = s$. Similarly, by definition $O^{-1}(s^d) = g$. Therefore the relation between g and s is identical to the relation between s^d and g. All operators are applicable in all states so all we need to do is to find the shortest sequence of operators that produces permutation O.

The PDB lookups are similar too. Let π be a *value-based* permutation ("renaming"), of s into g. That is, assume object a is in location $x \in s$. π specifies the name of the object for location $x \in g$. Now, since O and π are commutable then $s^d = O(g) = O(\pi(s)) = \pi(O(s)) = \pi(g)$. Hence, $h(s^d, g) = h(\pi(g), \pi(s)) = h(g, s)$. That means all heuristic lookups in DS for dual states s^d are equal to the reverse lookups in SFBDS and the two algorithms are equivalent.

SFBDS characteristics

SFBDS has the flexibility of deciding which side of the search to expand next. In fact, any gains achieved by SFBDS are solely determined by the quality of the jumping policy. In general, one wants to expand the side with the subtree below it that can be searched most efficiently. Three possible features for a jumping policy are considered here.

(1) Branching factor: For a node $N(x, y)$, x and y may

have different branching factors. Expand the state with the smallest branching factor. For example, consider a non-root 15-puzzle node $N(x, y)$ where state x has the blank in the center (branching factor of 3) while state y has the blank in the corner (branching factor of 1). Now consider a depth 5 tree with a non-uniform branching factor where the branching factors at depths $1 \ldots 5$ are fixed to 1, 4, 2, 2, and 2 respectively. If we change direction based only on the branching factor, we essentially get to 'skip' the worst branching factor in the tree. An optimal policy would switch directions at depth one, and expand 8 nodes total, while a forward search would expand 16 nodes and a reverse search would expand 32 nodes. The largest gain in this case is the ratio of the largest to smallest branching factor in the tree.

(2) Asymmetric heuristics: Assume that the graph is undirected and thus for every two states, x and y, $dist(x, y) = dist(y, x)$ where $dist(x, y)$ is the length of the shortest path between x and y. In many cases, admissible heuristics are symmetric too, meaning that $h(x, y) = h(y, x)$ (e.g., Manhattan distance). However, some admissible heuristics are not symmetric: $h(x, y) \neq h(y, x)$. An example is a goal-oriented PDB. Assume we build PDBs for state x (PDB_x) and state y (PDB_y). Each PDB relies on different features of the state, hence $PDB_x(y)$ likely stores a different heuristic value than $PDB_y(x)$. When an asymmetric heuristic exists we can perform these two possible lookups for node $N(x, y)$. As a first step, we can take the maximum of these two lookups as the heuristic for $N(x, y)$. The second step is more powerful. Assume that $h(x, y) > h(y, x)$, that expanding x will generate nodes (x_1, y) and (x_2, y) and that expanding y will generate nodes (x, y_1) and (x, y_2). Since the heuristic at x is larger, we expect that nodes (x_1, y) and (x_2, y) will have larger heuristics than (x, y_1) and (x, y_2). Thus, we choose to expand x. This was called the *jump if larger* policy (JIL) in (Zahavi *et al.* 2008).

(3) Side with larger heuristics: The preceding idea can be generalized. Even if the heuristic is symmetric we can do the following. Perform a 1-step lookahead and peek at all the children of x and measure their heuristic towards y. Similarly, perform a 1-step lookahead and peek at all the children of y and measure their heuristic towards x. If one side tends to have larger heuristic values, choose to expand that side. We refer to this as the $JIL(k)$ policy, where k is the lookahead depth. The JIL method described above is JIL(0).

Optimal jumping policies for IDA*:

The optimal jumping policy can be computed offline for IDA* under certain conditions. This gives us the minimum possible search effort that can be achieved. Let (x, y) be a task in the task tree. A jumping policy should decide whether to expand x or y. Our optimization algorithm performs the search below (x, y) for *both* cases until all solutions are found. We then backtrack up the tree and for each node choose the expansion with minimum search effort. This effectively doubles the branching factor. Hence, if a unidirectional search has complexity $O(b^d)$ finding the optimal policy has complexity $O((2b)^d)$. This is not feasible for long solution lengths d. The space needed for the optimal policy is $O(b^d)$.

Parent and duplicate pruning

Typically, when search algorithms expand a node N they do not generate the parent of N. This is usually done by keeping the operator that generated N and not applying its inverse to N. In SFBDS two operators are kept for $N(x, y)$, one for each of x and y. When a node is expanded from its forward (backward) side, the inverse of the operator that was used to first reach x (y) is not applied. In regular search the one exception is the root node. In SFBDS both the start and goal act as root nodes, slightly enlarging the tree.

DFS algorithms like IDA* do not perform duplicate detection (DD). If multiple paths exist to a node, that node and all its children may be expanded many times. Best-first algorithms like A* store open- and closed-lists performing DD. However, the DD problem in SFBDS is more complicated than in A*. As there are $O(V^2)$ possible tasks that can be created out of all possible pairs of states SFBDS has the potential to asymptotically increase the size of the search space. We devised a method to show when a particular task is guaranteed to be worse than similar existing tasks and can be pruned. We do not describe this pruning in detail here, as the gains of SFBDS in A* search are limited.

Experiments

SFBDS performance is demonstrated on the tile puzzles of size 15 and 24, the pancake puzzle, scale-free graphs and room maps. Puzzles have a relatively small and stable branching factor and belong to case 1 in the analysis section. However, heuristics can be used to diversify the sizes of the task trees induced by different jumping policies and relate them to the second case. Scale free graphs induce non-uniform, high branching factors and belong to case 2 of the domains. As an example of cases 3 and 4 we consider room maps as a path-finding problem.

On the puzzles, SFBDS is identical to DS. We repeated the experiments in (Zahavi *et al.* 2008), generating the same numbers. The JIL(k) results are new. For the non-puzzle domains duality does not exist and such searches are possible only due to our new SFBDS formalization.

15-puzzle with Manhattan Distance

To show the effectiveness of SFBDS on simple heuristics, we repeated the experiments first performed in (Korf 1985) using the Manhattan distance (MD) heuristic; this time including SFBDS with a number of jumping policies. The results are in Table 1 (top). The first line uses IDA* without any jumping and produces the identical node counts to those reported in (Korf 1985). The second line uses SFBDS-IDA* with the jumping policy of expanding the side with the smaller branching factor (BF). The branching factor is small, either 1, 2 or 3, (4 at the root) limiting the possible savings. Since the heuristic is symmetric, JIL(0) will never choose to reverse the search direction and is equivalent to regular IDA*. The next line reports the results for JIL(1) using the following jumping policy. Assume that T is the IDA* threshold, the current node is $N(x, y)$, and $f(N(x, y)) = k$. For the special case of the tile puzzle we know that the f-cost either remains the same or increases by

H	Alg.	Policy	Nodes	Time
15 puzzle				
MD	IDA*	Never	363,028,020	51s
MD	SFBDS	BF	256,819,013	37s
MD	SFBDS	JIL(1)	91,962,501	18s
MD	SFBDS	JIL(2)	71,290,100	17s
17 pancake				
regular	IDA*	Never	342,308,368,717	284,054s
reversed	IDA*	Never	14,387,002,121	12,485s
max	IDA*	Never	2,478,269,076	3,086s
max	SFBDS	JIL(0)	260,506,693	362s
max	SFBDS	JIL(1)	17,336,052	120s

Table 1: 15 puzzle (top). 17 pancake (bottom).

two. If $k = T$, then only count the children with $f = k$ (those with $f = k + 2$ will be pruned immediately); expand the side with the smaller count. If $k < T$ then all the children will be expanded and we want to estimate the number of nodes below this node with $f = k + 2$. We count the number of children but give a larger weight b (ideally b is the heuristic branching factor) to those with $f = k$ as they will generate a larger number of nodes with $f = k + 2$.

The results show the great potential in this direction. Even though the heuristic is symmetric, performing the JIL(1) policy reduced the number of generated nodes by a factor of 4 and the time overhead by almost a factor of 3. Further lookahead, JIL(2) provided modest gains.

Pancake puzzle with PDBs

A PDB is usually built to estimate the distance to a given goal state. However, in many permutation puzzles with the appropriate mapping of the tiles the same PDB can be used to estimate distances between any pairs of states. Therefore, given a node $N(x, y)$ and a PDB both $h_x(y)$ (*regular lookup*) as well as $h_y(x)$ (*reverse lookup*) can be calculated. Different PDB lookups are performed and different values can be obtained.

Table 1 (bottom) presents results averaged over 10 random instances of the 17-pancake puzzle. We used the same 7-token PDB used by (Zahavi *et al.* 2008) of the largest pancakes. The first line is a regular IDA* search with one PDB lookup. The second line always uses the reverse lookup. It produced inconsistent heuristic values because different tokens are being looked up at every step. Adding BPMX on top results in a 24-fold reduction in the number of nodes generated. Taking the maximum of both heuristics further improved the results. Line 4 shows the results of SFBDS with the JIL(0) policy where another 10-fold improvement was obtained. The first four lines already appeared in (Zahavi *et al.* 2008). However, we now also applied the new JIL(1) policy. With JIL(1), we get a further reduction by a factor of 15 in nodes, but only a factor of 3 in time because of the lookahead overhead. These are the state-of-the-art results for such PDBs on this domain. Similar tendencies were obtained for smaller sizes of this puzzle.

Optimal jumping policies for the pancake puzzle

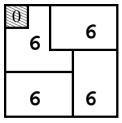
Table 2 shows the averaged results of the optimal jumping policy for the 10, 11 and 12 pancake puzzle using 1000 random instances each. Entries include the total number of nodes and the last iteration nodes (in brackets). We use the maximum of the regular and reverse lookup using the PDB heuristic of the 7 largest tokens. BPMX is disabled. The last row shows the minimal possible search effort for SFBDS in the last iteration with an optimal jumping policy. Even though our jumping policies perform well compared to unidirectional search, the results suggest that there is potential for further improvements. Note that JIL is a generic jumping policy. Hence, by incorporating domain-dependent knowledge it is possible to build more sophisticated policies.

pancakes	10	11	12
avg. sol.	8.683	9.66	10.699
policy	nodes generated (last iteration)		
regular	83 (68)	405 (286)	3,728 (2,538)
JIL(0)	73 (60)	307 (214)	2,247 (1,482)
JIL(1)	65 (54)	233 (165)	1,670 (1,108)
optimal	(43)	(94)	(458)

Table 2: Optimal policy on the pancake puzzle

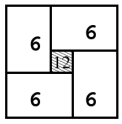
#	Heuristic	Policy	Nodes
1	r,r*	-	43,454,810,045
2	r,r*,rev,rev*	-	13,549,943,868
3	r,r*,rev,rev*	JIL(0)	3,948,614,947
4	r,r*,rev,rev*	JIL(1)	1,778,435,449

(a) SFBDS on the 24-puzzle.



Alg.	Octile		Differential	
	nodes	win	nodes	win
A*	**7,835**	N/A	1,179	N/A
BF	329,837	**12.5**%	4,165	18.2%
JIL(1)	804,426	1.6%	4,847	42.4%
DW	34,037	5.0%	**841**	**67.0**%

(c) SFBDS with A* on room maps.



(b) PDBs

Table 3: Results on the 24-puzzle and room maps.

24-puzzle

For the 24-puzzle, the same $6 - 6 - 6 - 6$ PDB partitioning from (Korf & Felner 2002) was used. If we only use the traditional goal state, then the PDB of Table 3b (top) is used as it has the blank in the corner. In (Zahavi *et al.* 2008) they showed that 8 6-tile PDBs are enough to be able to perform a 6-6-6-6 partitioning towards any possible blank location. For example, Table 3b (bottom) shows how four 6-tile PDBs can be used when the blank is in the center.

In (Korf & Felner 2002) 50 random instances were optimally solved. Following (Zahavi *et al.* 2008) we only report results on the 25 instances with the shortest optimal solution in Table 3a. The first three lines are identical to those in (Zahavi *et al.* 2008); the fourth line is new. Line 1 presents the benchmark results from (Korf & Felner 2002) where the maximum between the regular PDB (r) and its reflection about the main diagonal (r^*) were taken. Line 2 is the case where the maximum between all possible four PDB lookups were used (*regular*, *reversed* and their two reflections about the main diagonal). Line 3 shows SFBDS which used a policy based on JIL(0) (called J24 in (Zahavi *et al.*

Algorithm	BFS	BDBFS	DFID	SFDFID
Nodes	16,427	212	824,682	18,056
Time	35s	3ms	936ms	17ms
Nodes - 10%	56,154	1,775	-	21,142
Time - 10%	50s	18ms	-	21ms

Table 4: Results on a scale-free graph.

2008)). The last line shows the new JIL(1) results where a further reduction of a factor of 2.2 is obtained.

Scale-Free Graphs

As an example for domain of case 2 of the analysis section we investigate *scale-free graphs* (or *power-law graphs*) (Faloutsos, Faloutsos, & Faloutsos 1999). We used the R-MAT (Chakrabarti, Zhan, & Faloutsos 2004) algorithm to generate a scale-free graph with 100,000 nodes and 400,000 edges. The branching factor ranges from 1 to 150.

We report the results for one such graph in Table 4, although other graphs gave similar results. No default heuristic is available in these graphs, so we consider the following algorithms. *Breadth-first Search* (BFS) finds optimal paths by expanding nodes, best-first, based on their g-value (=depth). Bidirectional breadth-first search (BDBFS) expands nodes from the start and goal simultaneously, stopping when the frontiers meet. DFID is a depth-first search with iterative deepening cost limits. SFDFID is a single-frontier bidirectional version of DFID with a policy that expands the side with the lower branching factor.

The results are averaged over 96 instances (out of 100) that could be solved by all algorithms. The bottom two lines are averages over the hardest 10 problems only. SFDFID expands 46 times fewer nodes than DFID and is 55 times faster. Over all problems, BDBFS expands the fewest nodes and is fastest. BFS and BDBFS are very slow due to data structure overheads. Although SFDFID expands 85 times more nodes than BDBFS, it is only 5.7 times slower, and on the hardest problems it is actually comparable to BDBFS as its constant time per node is much smaller. A custom implementation of SFDFID for this domain would likely be even faster, as the branching factor in each direction can be cached, an optimization we did not perform.

These results illustrate the gains that are possible in domains with a variable branching factor. SFDFID outperforms regular DFID, and has comparable performance with BDBFS, yet only uses memory linear in the solution depth.

SFBDS-A* on room maps

Room maps are structured by a grid of rooms with random doors between them. This domain is an example of case 3 given in the analysis section. As A* keeps a closed-list, searching to the edge or a corner of a room is comparable to a dead end in the search tree because the goal can only be reached by backtracking through previously expanded states.

We performed experiments on 24 maps, each having 32x32 rooms of size 7x7 and random paths that connect the rooms. Our test set consists of 11,205 pairs of start and goal states.We experimented with the trivial *octile distance* heuristic and with the more informed memory-based *differential heuristics* (DH) (Sturtevant *et al.* 2009) with ten canonical states.

The results are shown in Table 3c. A vertex with branching factor two is called a *doorway*. The doorway policy (DW) only reverses the search direction (jumps) at such states, i.e., if for node $N(x, y)$ x or y are doorways, search towards it. We measure the number of nodes expanded (nodes), averaged over the test set, and the number of instances where the respective algorithm expands fewer nodes than A* (win) as a percentage of the total number of instances. The respective best results are highlighted. The first line shows the results for regular A*.

SFBDS-A* with the octile heuristic and the branching factor policy performs poorly, rarely beating A* and expanding 40 times more nodes on average. This is due the potential V^2 blow-up in tasks versus states, particularly because the octile heuristic is weak.

With good heuristic guidance the algorithm can avoid expanding states that lead to dead ends. Therefore, SFBDS with differential heuristic and doorway policy expands fewer nodes than regular A* on average. Our algorithm performs better in more than 67% of the test instances and expands 1.4 times fewer nodes. These results suggest that the gains from SFBDS will be minimal in a highly connected graph unless we can take advantage of special properties of a domain.

Conclusions

SFBDS is a general approach for bidirectional search which is especially applicable to depth-first search. It is far simpler and more general than the previous dual search ideas. We provide new analysis that provides deeper insight into when such an approach will work and what properties a domain must have to benefit from a SFBDS approach.

Acknowledgments

This research was supported by the Israeli Science Foundation (ISF) grants No. 728/06 and 305/09, and by iCORE.

References

Chakrabarti, D.; Zhan, Y.; and Faloutsos, C. 2004. R-mat: A recursive model for graph mining. In *SDM*.

Faloutsos, M.; Faloutsos, P.; and Faloutsos, C. 1999. On power-law relationships of the internet topology. In *SIGCOMM*, 251–262.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI-05*, 103–108.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *JAIR* 7:283–317.

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.

Sturtevant, N.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI-09*, 609–614.

Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artif. Intell.* 172(4-5):514–540.