Single-Frontier Bidirectional Search

Carsten Moldenhauer
Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin, Germany
moldenha@informatik.hu-berlin.de

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

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

Abstract

We introduce a new bidirectional search algorithm, *Single-Frontier Bidirectional Search* (SFBDS). Unlike traditional BDS which keeps two frontiers, SFBDS uses a single frontier. 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. We provide theoretical analysis that explains when SFBDS will work validated by experimental results.[1]

Single-frontier bidirectional search

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.

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. 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 (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.

Our new algorithm is called *single-frontier bidirectional search* (SFBDS). 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. 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. 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.

[1]A full version of this works appears in (Felner et al. 2010)

Solutions or cost estimates from node N are naturally passed up to the parent of N, regardless of the direction used for N. 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.

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 $N(s, g)$ as the root of the tree. Any admissible algorithm can be used to search for a shortest path from the root to any goal node in the task search tree.

The concept of *duality* and an algorithm called *dual search* (DS) (Zahavi et al. 2008) was introduced in the context of permutation state spaces. 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. SFBDS generalizes DS to all possible state spaces and is simpler to understand.

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 two general types of domains for the effectiveness of SFBDS:

Case 1: Exponential domains with uniform branching factor b where IDA* is traditionally used. Since the branching factor is uniform all the task trees have the same structure and they will all have O(b^d) node expansions. In such domains, specific jumping policies exploiting heuristic information have the potential to reduce the search effort.

Case 2: Polynomial domains where A* is traditionally used. In such case, usually the graphs have many cycles. Best-first algorithms like A* store open- and closed-lists and perform duplicate detection (DD). However, performing DD in SFBDS is more complicated than in A*. If there are V states in the graph, there are $O(V^2)$ possible tasks that can be created out of all possible pairs of states and the chance for identifying a duplicate node is much smaller than unidirectional search. Therefore SFBDS has the potential to asymptotically increase the size of the search space. Furthermore, unlike exponential domains, polynomial domains might have deadends. SFBDS with might not recognize dead ends until at least one of the two states of a node is a dead end. This has the potential to significantly increase the search effort.

Jumping policies

SFBDS has the flexibility of deciding which side of the search to expand next by choosing an appropriate 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.

(2) Side with larger heuristics: Assume that the graph is undirected and thus for every two states, x and y, $dist(x, y) = dist(y, x)$. 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. When an asymmetric heuristic exists we can perform these two possible lookups for node $N(x, y)$. If $h(x, y) > h(y, x)$ we choose to expand x and vise versa. This was called the *jump if larger* policy (JIL) in (Zahavi et al. 2008). Even if the heuristic is symmetric we can do the following. Perform a 1-step lookahead and peek at all the children of x (y) and measure their heuristic towards y (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).

Experiments

Puzzles have a small and stable branching factor and belong to case 1 above. We demonstrate the effectiveness of SFBDS on the 15 puzzle and on the pancake puzzle. We repeated the experiments first performed in (Korf 1985) using the simple 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 next line reports the results for JIL(1). 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.

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).

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.

We experimented with *scale-free graphs* and with grids which are commonly used for pathfinding as examples for polynomial domains where A* is used. Since dead ends occur and the graph is highly connected our results suggest that the gains from SFBDS are minimal for these domains.

References

Felner, A.; Moldenhauer, C.; Sturtevant, N.; and Schaeffer, J. 2010. Single frontier bidirectional search. In *AAAI-10*.

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

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.