Efficient Single Frontier Bidirectional Search

Marco Lippi
Dip. Ingegneria dell'Informazione
Università di Siena
Italy
lippi@dii.unisi.it

Marco Ernandes
QuestIT s.r.l. and
Dip. Ingegneria dell'Informazione
Università di Siena
Italy
marcoernandes@gmail.com

Ariel Felner
Dept. Information Systems Engineering
Ben-Gurion University
Israel
felner@bgu.ac.il

Abstract

The Single Frontier Bi-Directional Search (SBS) framework was recently introduced. A node in SBS corresponds to a pair of states, one from each of the frontiers and it uses front-to-front heuristics. In this paper we present an enhanced version of SBS, called eSBS, where pruning and caching techniques are applied, which significantly reduce both time and memory needs of SBS. We then present a hybrid of eSBS and IDA* which potentially uses only the square root of the memory required by A* but enables to prune many nodes that IDA* would generate. Experimental results show the benefit of our new approaches on a number of domains.

Introduction

Bidirectional search is a well known idea but is not used very often, especially when the search is guided by a heuristic function (h-function) and when the optimal solution is required. The main reason, well studied by (Kaindl and Kainz 1997), is the *meet in the middle* problem, which is the problem of guaranteeing the optimality of the solution when the two search frontiers meet. Recently, a new bidirectional search framework called Single Frontier Bidirectional Search (SBS) was introduced (Felner et al. 2010)[1]. The main idea of SBS is to search through a *double node* search tree, where each *double node* (from now on, simply *node*) N includes two states, one state s from the forward search and one state g from the backward search. The task in node N is to find the shortest path between s and g. This task is recursively decomposed by expanding either s or g and generating new nodes between (1) s and the neighbors of g, or (2) g and the neighbors of 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 jumping policy, a tree is induced which can be searched using any admissible search algorithm such as A* (SBS-A*) or IDA* (SBS-IDA*). The original SBS paper (Felner et al. 2010) studied different jumping policies and their influence on the performance; impressive savings were obtained in many domains, but in some cases a blowup in the search tree made SBS ineffective.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] It was labeled by the authors as SFBDS but to improve the readability we shorten the notation and denote it by SBS.

In this paper we take SBS several steps further. First, we provide deeper insights on the implementation of this algorithm and introduce four interdependent enhancements that can be implemented within the SBS framework to get significant improvements in both space and time complexity. The main idea is to split the memory into two data structures. The first is the OPEN-list which stores nodes but in the form of two pointers, one for each of the states. The second is a *state transposition table* which stores states that were already seen during the search either in the forward or backward directions. This split may save a large amount of memory and enables an effective duplicate pruning mechanism for SBS. SBS with our enhancements is called Enhanced-SBS (denoted eSBS and when coupled with A* denoted by eSBS-A*).

In addition, based on these enhancements, we present a new variant of SBS, called eSBS-H which is a hybrid between eSBS-A* and SBS-IDA*. Similarly to eSBS-A*, eSBS-H uses the state transposition tables for effective duplicate pruning, but similarly to IDA*, it does not use an OPEN list and the search is performed by a series of depth-first search iterations on nodes. While A* needs memory linear in the number of generated nodes whereas IDA* does not need any memory, eSBS-H offers a good compromise and its memory needs is the square root of the number of generated nodes. While the number of generated nodes is quadratic in the number of states ($O(b^{2d})$), unlike ordinary bidirectional searches eSBS-H needs memory which is only linear in the number of states seen ($O(b^2)$). Experimental results show that eSBS-A* outperforms basic SBS-A* and that eSBS-H outperforms both eSBS-A* and SBS-IDA* on a number of domains including the 15 puzzle, the pancake puzzle and room maps.

Single-frontier bidirectional search (SBS)

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., x) indicate states (or vertices) of the input graph.

Assume the task is to find a path between s and g on an undirected graph. Traditional unidirectional 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

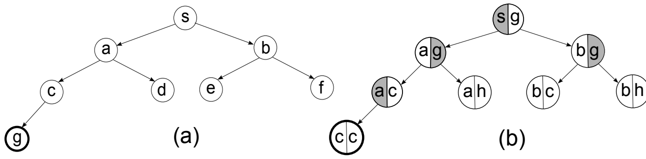

Figure 1: Unidirectional vs. SBS search trees (expanded states are represented in gray for SBS).

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.

SBS: formal definition

The search tree of SBS (Felner et al. 2010) uses *double nodes*. Each double node is defined as a pair of states x and y and is denoted by $N(x, y)$. The root node $R(s, g)$ includes the start state s and the goal state g. The task beneath a node $N(x, y)$ is to find a shortest path between x and y. This can be done by either treating x as the start and y as the goal, searching from x to y or by *reversing* 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 $N(x, y)$ 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. We employ a *jumping policy* to decide which side should be expanded for every given node. The search terminates when a *goal node* is reached, where a goal node is in the form $N(x, x)$. The choice made by the jumping policy for the search direction in N is reflected by N's children only, but no other node that is currently in the search tree 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.

Examples

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

Now, consider searching the same tree with SBS (Figure 1b). Nodes are labeled with the shortest-path task that should be solved beneath them. Within each node, the state expanded is shaded in the figure where left (right) means expanding the forward (backward) state. The jumping policy used in this example is the *alternate policy*, so that at even depths the forward state is expanded and at odd depth the backward state is expanded. For example, at the root, the task is (s, g) resulting in two children, (a, g) and (b, g). At node (a, g), however, the jumping policy chooses g for expansion. This generates nodes for all neighbors of g, leading to (a, c) in our example. Finally, at (a, c), state c is chosen for expansion, generating a goal node (c, c).

Edges in the SBS 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). 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 backward expansions are appended to the end of the path. Thus, the path of (s, a, c, g) is constructed from this branch.

Jumping policies

We use a *jumping policy* to choose which direction to continue the search at each node[2]. Trivial policies are the *never jump* policy, replicating a classical unidirectional algorithm, where we always expand the forward side. Similarly, The *jump only at root* policy, corresponds to a unidirectional backward search form the goal to the start where we always expand the backward side. A deep study on jumping policies was performed by (Felner et al. 2010): the authors concluded that it would be best to devise a specific jumping policy for each possible domain, but also suggested good general purpose policies (Felner et al. 2010). Among them is the *branching factor* (BF) policy, which expands the side with the smaller branching factor. Similarly, in the case of asymmetric heuristic the *jump if larger* policy for node $N(x, y)$ chooses to expand x (y) if $h(x, y) > h(y, x)$ (and vice versa). Another jumping policy that we introduce here is called the *alternating policy*, or *always jump*, which at the odd levels expands the forward side and at the even level expands the backward side.

Figure 2 shows all these policies in a *policy space tree*. Nodes of this tree correspond to the different order of expansion actions. At each step, moving left corresponds to a forward expansion and moving right corresponds to a backward expansion. Each path from the root of this tree to a leaf of level d corresponds to a given jumping policy. The three simple policies described above are shown in this tree. Regular unidirectional search (never jump) is to always go left. Backwards search (jump only at root) is to always go right. The alternate policy is shown in the middle were the left and the right children are taken alternately.

SBS should be seen as a general paradigm. On top of SBS we should first specify the *jumping policy* which decides which side to expand for each of the nodes. Given such a policy, a search tree is induced. At any node $N(x, y)$ the remaining search effort can be estimated by a *front-to-front* heuristic function, $h(x, y)$, estimating the distance between x and y. This search tree can then can be searched using any search algorithm such as breadth-first search, A*, IDA* etc. Such algorithms are labeled by SBS-A*, SBS-IDA* etc.

[2]The term *jump* comes for an earlier version of SBS called *dual search* (Zahavi et al. 2008) where changing the direction of the search resulted in *jumping* to the so called *dual state*. Here we do not actually *jump* as the two states are at hand. The policy needs to decide which of these to expand and may *jump* from side to side.

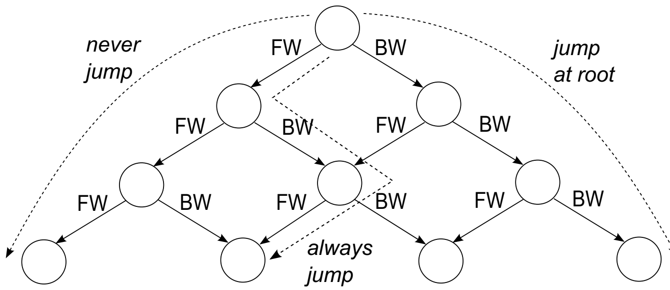

Figure 2: Some examples of jumping policy: *always jump* (always expand FW direction), *jump at root* (always expand BW direction), *always jump* (alternately expand FW and BW directions).

The optimality of the solution is naturally guaranteed if an A*-based search is activated on such a tree, employing an admissible front-to-front heuristic function.

Enhanced SBS

In this section we present our enhanced-SBS-A* (labeled eSBS-A*). eSBS-A* is based on four interdependent enhancements to basic SBS-A* (as described in (Felner et al. 2010)) which greatly speed up the search. We describe each of them in turn in the following subsections. Based on these enhancements, we will then introduce our new algorithm eSBS-H (a hybrid between eSBS-A* and SBS-IDA*).

Referencing states with pointers

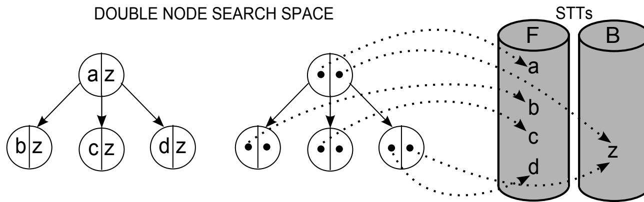

Figure 3: Representation of nodes in classic SBS (left) and in eSBS (right), using pointers to the states stored in the STTs.

A basic implementation of SBS-A* explicitly stores two states inside each node (see Figure 3.left). The key ingredient of eSBS-A* is based on the simple observation that the number of states seen during the execution of SBS is greatly smaller than the number of generated nodes, since the same state will be contained in many different nodes. The number of nodes grows quadratically with the number of states as each pair of states from the two frontiers, respectively, can potentially have their own node. Therefore, the first enhancement is not to maintain states inside nodes, but just their references through the use of pointers. This idea is shown in Figure 3.right. Forward and backward states that are seen during the search states are stored inside two *state transposition tables* (STT), named *Forward STT* (FSTT) and *Backward STT* (BSTT), respectively. Nodes of the SBS-A* search tree are stored in an OPEN list but such nodes simply contain two pointers, one to a state in the FSST and one to a state in BSST.

There is an immediate memory benefit in domains where a state description is larger than the size of a pointer. However, the main purpose of storing state pointers is to obtain further speedup and pruning via our pruning and caching techniques, described below. These cannot work with a basic implementation of SBS-A*.

Parent pointers and $best\text{-}g$ update

An additional reduction in the number of stored pointers can be obtained by moving parent pointers from nodes to states within the STT. In this way, each state will contain a pointer to its predecessor. Thus, while the same state occurs many times inside nodes, there will be only one parent pointer for each state. This might cause a quadratic reduction in the number of pointers. These parent pointers will be used to reconstruct the solution path once the algorithm has terminated successfully.

Using such parent pointers introduces a problem as parent pointers are no longer unique to a given node: two nodes sharing the same state will now share the same parent pointer. To guarantee admissibility we must ensure that the pointed parent of each state lies on the optimal path. This can be obtained by providing each state with a variable, called $best\text{-}g$, that keeps track of the lowest observed cost, from the corresponding first state (start or goal, depending on the direction) to the state itself. The parent pointer will be updated whenever $best\text{-}g$ is updated as follows. When generating a new node, it is necessary to check whether each of the two states can be found in the appropriate STT. If a state had not yet been seen, it would be saved in the respective STT and its $best\text{-}g$ would be initialized to the total cost of actions that originated it. If the state was already in the STT, it must be checked whether its new g-value is smaller than the previously recorded $best\text{-}g$. In such case, the state must be updated by resetting the $best\text{-}g$ and the pointer to the parent state.

Duplicate detection and best-g 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). As just explained, in eSBS two operators (in the form of pointers) 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.

Furthermore, best-first algorithms like A* store OPEN and CLOSED lists and usually perform duplicate-detection (DD) as follows. When a node is generated, it is searched over both the OPEN and CLOSED lists: if the same node already exists in these lists, we keep the copy with the smallest g-value and prune the other node[3]. In SBS, a *trivial duplicate detection* technique would be to check whether the

[3]Strictly speaking, if the heuristic is *consistent* then only nodes in OPEN should be checked because nodes that were already expanded and moved to CLOSED are guaranteed to already have the best g-value. If, however, the heuristic is inconsistent both OPEN and CLOSED should be checked and nodes from CLOSED might be re-opened (Felner et al. 2011).

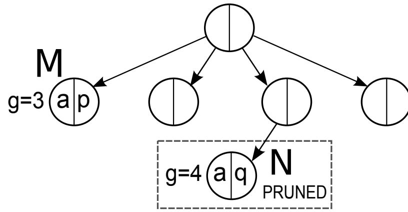

Figure 4: An example of $best\text{-}g$ pruning.

same node already exists in OPEN or CLOSED. However, since there are $O(V^2)$ possible nodes that can be created out of all possible pairs of states, this is not enough.

An improved DD technique which we call *$best\text{-}g$ pruning* is as follows. During node generation, whenever the g value of an already seen state a (both in the forward or backward direction) is greater than its current $best\text{-}g$ value, we know that this node lies on a suboptimal path and all its descendants would fail the $best\text{-}g$ check anyway: thus, this node can be pruned from OPEN. In other words, duplicate pruning is done for each frontier of the search on its own. But, these pruning have mutual effects as duplicates states will never be added to a new generated node. This technique is shown in Figure 4. State a was first seen in node $M = (a, p)$ with $g = 3$: when seen later inside node $N = (a, q)$ with $g = 4$, node N can be pruned since it necessarily lies on a suboptimal path. Within eSBS-A* search, including in the OPEN set such a suboptimal successor node, obviously does not lead to a final optimal solution, but it could lead to a great waste of computational resources.

Successor caching

The improvements which have been presented so far rely on the fact that the number of generated nodes greatly dominates the number of states stored in the STTs. This observation implies that the same state can appear in many different nodes, and thus the successor function may be executed multiple times on the same state. We exploit this condition by adding to each state in the STT a list of pointers to its successor states. When a state s is first generated, its successor pointers are set to NIL. The first time the successor function is called on s (when a node including s is expanded) we update all the successor pointers. Before calling the successor function we check whether the successors are already available, in which case we simply return the cached pointers. This technique, that we call *successor caching* has a strong contribution in reducing the number of executions of the successor function. This greatly reduces the resources required to generate the successors, which might be costly in a real-world problem. For example, successor states could be obtained by calling a third-party web service (e.g. a Google Maps API call) that may require important financial resources (e.g. subscriptions, bandwidth). On the other hand, caching successors increases memory consumption: since we add a pointer for each successor, the overhead is proportional to the branching factor of the problem. Hence, each domain introduces a different memory vs. speed tradeoff, and the use of successor caching should be evaluated beforehand.

Our resulting eSBS-A* algorithm is therefore SBS-A* coupled with all the enhancements described above: state referencing, parent pointers, $best\text{-}g$ pruning and successor caching. These enhancements enable a large speedup in the search as well as a reduction in the amount of memory.

eSBS-A* lite

The $best\text{-}g$ pruning technique detects and prunes those duplicate states within each frontier where one copy of a given state had a smaller g-value than the other copy. But, if both copies had the same g-value, the resulting node must be generated. Therefore, the same node (with the same g-value) can be generated via different paths. When keeping OPEN and CLOSED lists of nodes, the traditional trivial node pruning can detect such duplicate nodes.

However, with eSBS-A*, the use of $best\text{-}g$ pruning technique might already detect many duplicate nodes and only few of them are pruned by the trivial duplicate pruning. For this reason, we propose also a special version of eSBS-A*, named *lite*, which does not use a CLOSED list at all, and moreover does not perform the trivial duplicate detection in the OPEN list, but only relies on $best\text{-}g$ evaluations to prune the search. The experimental results will confirm the effectiveness of this variant, which produces an algorithm that generates more nodes than plain eSBS-A*, but is faster and uses less memory. The optimality of this *lite* algorithm can be easily proved as well.

eSBS-H: a hybrid between A* and IDA*

The enhancements we described above were developed for SBS-A*, but they might be interleaved with other algorithms too. We now present a new algorithm, called eSBS-H, which is a hybrid between eSBS-A* and SBS-IDA*.[4] eSBS-H exploits all the computational improvements of eSBS and results to be a very smart compromise between computational speed and memory consumption.

Like SBS-IDA*, eSBS-H does not employ an OPEN list to decide which node to expand next and it uses an iterative deepening approach on the double node search tree. However, like eSBS-A*, eSBS-H stores and maintains the STTs for the two search frontiers. Thus, all the above methods, parent pointers, best-g pruning and successor caching can be still performed within eSBS-H. In fact, eSBS-H acts like IDA* with transposition table, but its transpositions are not in the form of nodes but in the form of states. eSBS-H can also be seen as a further step of eSBS-A* *lite*. While eSBS-A* *lite* does not use the CLOSED list, in eSBS-H both the CLOSED and OPEN lists are not used and the search proceeds according to the IDA* approach, but, instead of pushing/popping nodes in a stack which is newly re-generated on-the-fly at each iteration, it stores the forward/backward states in two STTs, as in eSBS-A*. In this way, by exploiting the successor caching technique, at each new iteration of the algorithm at depth d, the whole subtree of depth $d-1$ will be

[4] In fact, it can be seen as the iterative deepening implementation of eSBS (thus eSBS-IDA*), but we prefer to call it a *hybrid* to highlight the fact that it combines features of both A* and IDA*.

immediately available and would not need to be generated, but just accessed via the successor pointers. Furthermore, g-pruning is also available. Thus, while the search strategy is that of iterative deepening, eSBS-H exploits many of the benefits of keeping the STTs.

Complexity issues

We now compare the time and space complexities of the different algorithms. The number of search nodes generated by eSBS-A* and SBS-IDA* grows exponentially with the depth of the solution. Similarly, we can observe from Figure 3 that eSBS-H will generate b^k nodes at each level k and will terminate at a search depth of d, hence the number of generated nodes for eSBS-H is: $O(\sum_{k=1}^{d} b^k) = O(b^d)$.

The time complexity depends mostly on the number of generated nodes, thus for all optimal heuristic search algorithm, this complexity is $O(b^d)$. Of course, in practice SBS-IDA* will generate the largest number of nodes as it does not perform any kind of duplicate detection and it generates many nodes more than once in the different iterative deepening iterations. On the other hand, eSBS-A* generates every node at most once and performs full duplicate detection, and it will thus generate the least amount of nodes. eSBS-H is a hybrid: it performs the $best$-g duplicate pruning but generates nodes in an iterative deepening manner. Different constants are associated with each of these algorithms and as we will see below, eSBS-H has the best actual CPU time. In addition, the caching technique described in Section greatly contributes to speed up the computation, as the experimental section will highlight.

Now, let us consider the memory complexity. Recall that eSBS-A* stores both the OPEN and CLOSED lists as well as the STTs while eSBS-H only stores the STTs. Assume we employ the *alternate* jumping policy (which expands the forward state at one level and the backward at the following one). Only those states corresponding to the successors of either the forward or the backward states will be generated in the relevant STT, heading to:

`Level 0` $\rightarrow$ 2 new states
`Level 1` $\rightarrow$ b new states (fw expansion)
`Level 2` $\rightarrow$ b new states (bw expansion)
`Level 3` $\rightarrow$ b $\times$ b new states (fw expansion)
`Level 4` $\rightarrow$ b $\times$ b new states (bw expansion)
...
`Level` d-`1` $\rightarrow \underbrace{b \times b \times \ldots \times b}_{d/2\ times}$ new states (fw expansion)
`Level` $d \rightarrow \underbrace{b \times b \times \ldots \times b}_{d/2\ times}$ new states (bw expansion)

Thus, the overall number of stored states in the STTs will be:

$$O(\sum_{k=1}^{d/2}(b^k + b^k)) = O(2\sum_{k=1}^{d/2} b^k) = O(b^{d/2})$$

since the search process will store $2 \cdot b^k$ states every two levels[5]. Assume that c_p is the constant memory allocated for a pointer and c_s is the constant memory needed to store a state. While c_s is mainly problem-dependent, c_p depends primarily on the hardware/software implementation. If $c_s \leq c_p$ eSBS-H will not reduce memory consumption, since the overhead of maintaining pointers to the states would be greater than the advantage of reducing the number of stored states. Anyway, in most problems we can expect c_s being much greater than c_p, heading to considerable memory savings.

Algorithm	Nodes	States	Total
eSBS-A*	$2 \cdot c_p\ b^d$	$c_s \cdot 2 \cdot b^{d/2}$	b^d
eSBS-H	$2 \cdot c_p\ d$	$c_s \cdot 2 \cdot b^{d/2}$	$b^{d/2}$
SBS-IDA*	$2 \cdot c_p\ d$	-	d

Table 1: Memory complexity comparison.

The space complexity of the different algorithms is summarized in Table 1 . The first column corresponds to the the number of stored nodes. eSBS-H and SBS-IDA* are both based on depth-first search and thus only need memory linear in d. By contrast, eSBS-A* needs memory exponential in d. The second column presents the number of stored states and the third column is the total asymptotic complexity. Given modern memory capacities, SBS-IDA* essentially needs almost no memory, and thus does not exploit the available memory. By contrast, eSBS-A* needs memory which is exponential in the depth of the search and this may be beyond the abilities even of modern machines. Clearly, eSBS-H offers a nice compromise and potentially only needs memory which is the square root of the memory needs for eSBS-A*.

Anyhow, it should be underlined that the number of generated nodes and stored states, and therefore the ratio between the memory requirements of different algorithms, will highly depend, on the adopted jumping policy and more importantly on the heuristic function guiding the search. We can predict that with no heuristic information the number of states can be effectively estimated as twice the square root of the number of generated nodes. With increasing heuristic information this ratio will grow, slightly reducing the efficiency of the proposed framework, and in the trivial case of domains with perfect information, the number of states will match the number of nodes. This hypothesis can be easily proved by running eSBS-A* and eSBS-H with increasing heuristic quality. We have done this for the 8-puzzle. With $h = 0$ (no heuristic information), the ratio between the number of states and the square root of number of nodes generated by eSBS-A* is 3.3, whereas with the misplaced tiles heuristic this grows to 7.3 and with Manhattan distance it reaches 12.7. This data show that for a weaker heuristic we are much closer to the square root bound. Similarly, with eSBS-H these three ratios correspond to 2.4, 5.4 and 12.2, respectively.

Another feature that can strongly affect the states-vs-

[5]More precisely, the number of new states would be $b_{fw}^k + b_{bw}^k$, if the branching factors along the two directions were different.

nodes ratio is the jumping policy. It is easy to prove that the square root ratio could be guaranteed only with a perfectly balanced policy (as the *alternate* policy), where both directions are equally explored. On the other hand, with a never-jump policy (which reproduces a single-direction search) the number of states would match the number of nodes and, thus, eSBS would provide no added value. Any other jumping policy would fall between these two extremes.

The experimental results presented in the forthcoming section will confirm the intuition that the number of different states that an eSBS search encounters is substantially smaller than the number of visited double nodes. This observation is crucial to understand the properties of the model and explain its capability to improve heuristic search. As a consequence, the more information will be stored in the state rather than in the node, the greater will be the gain of our new approach.

Experiments

We performed experiments on the Fifteen puzzle, the Pancake puzzle and on the Room Maps data set. All tests were run on a 3GHz processor with 4MB cache, and 16GB of RAM. We want to point out that, for each considered algorithm, we used the same implementation in all the tested domains, just adapting the state representation and the heuristic function computation. Hence, we employed a very general implementation with no domain-dependent tricks to be more efficient. Nevertheless, for this reason, some algorithms might be slower in CPU time if compared to ad-hoc implementations with domain dependent enhancements for some of the considered domains.

Fifteen puzzle

In our first domain, we experimented on the 100 random instances of the 15 puzzle first used in (Korf 1985). We compared the following algorithms: unidirectional A*, SBS-A*, SBS-IDA*, eSBS-A*, eSBS-A* *lite*, unidirectional IDA*, SBS-IDA*, eSBS-H and also unidirectional IDA* with a transposition table (IDA$^*_{tt}$). For each problem instance we recorded the number of states stored by eSBS-H in the two STTs and then allowed (IDA$^*_{tt}$) to store exactly the same number of states. The intent of implementing IDA$^*_{tt}$) is to compare our framework with standard transposition tables, with equal memory requirements. We employed the *always jump* policy (labeled A), but also tested the impact of the branching factor policy (BF) (choose to expand the side with expansion the lower BF) and, additionally, we also tried an enhanced version which uses the BF policy but in cases of a tie it uses an alternate policy (this is labeled BF/A). Table 2 summarizes the performance of the different algorithms with the Manhattan distance heuristic. The results are averaged over the 83 configurations solved by all the algorithms (top) and for those who could also solve all 100 instances (bottom) where IDA* exactly replicate those reported in (Korf 1985). SBS-A* only solved 77 configurations out of 100, running out of memory on the remaining. Thus, for SBS-A* the results on the 83 configurations are based only on these 77 instances. In practice, for the entire 83 configurations the numbers are a little higher.

Algorithm	Policy	Nodes	States	Time
83 instances				
A*		7,911,967	7,911,967	666,259
SBS-A*	A	14,758,988	14,758,988	245,609
eSBS-A*	A	16,162,187	250,278	114,453
eSBS-A* *lite*	A	21,046,781	250,379	78,386
IDA*		88,470,043	–	70,051
IDA$^*_{tt}$		87,703,982	355,371	142,711
SFBDS-IDA*	A	63,391,897	–	90,491
SFBDS-IDA*	BF	40,270,972	–	60,333
eSBS-H	A	46,208,977	355,371	16,896
eSBS-H	BF	17,165,052	4,237,163	25,297
eSBS-H	BF/A	28,057,228	535,756	12,010
100 instances				
IDA*		363,028,080	–	287,770
IDA$^*_{tt}$		358,775,931	742,963	633,094
SFBDS-IDA*	A	239,252,329	–	342,321
eSBS-H	A	164,027,089	742,963	58,559
eSBS-H	BF/A	92,339,888	1,179,106	38,078

Table 2: Results on the 15-puzzle. Time is measured in ms.

The experimental results clearly show the benefits of the eSBS variants. The first four lines compare four A* versions: unidirectional A*, SBS-A* and the eSBS-A* variants. The results on the A* versions show that all SBS versions generated more nodes than unidirectional A*. This phenomenon was well studied in (Felner et al. 2010; Zahavi et al. 2008) and is due to the *penalty of jumping* (Zahavi et al. 2008) and the task blowup (Felner et al. 2010). However, clearly, even though more nodes were generated by eSBS-A* and eSBS-A* *lite* the number of states stored was significantly smaller and the total CPU time was reduced by up to a full order of magnitude. The next three lines compare IDA* to SBS-IDA*, replicating the results shown by (Felner et al. 2010). SBS-IDA* generates a factor of 2 less nodes and much of it reflects in the CPU time. The three subsequent lines show the behavior of eSBS-H with the different policies, on the 83 easy configurations. eSBS-H is up to 6 times faster than unidirectional IDA* and up to 5 times faster than SBS-IDA* at the cost of slightly larger memory needs due to storing the STTs. It only needs to store 535,756 states compared to 16,162,187 nodes plus 250,278 that A* stores. The efficiency of eSBS-H increases when we report results on all 100 instances. For the best jumping policy, eSBS-H generates a factor of 4 less nodes than IDA* and the running time is faster by a factor of 7.5.

It is very interesting to notice that, when coupling the algorithm with a smarter jumping policy, eSBS-H stores a larger number of states while generating a smaller number of nodes: this happens because using the BF policy we expand the forward state if $b_{fw} \leq b_{bw}$, and therefore the many tie-breaks between b_{fw} and b_{bw} are all resolved in favor of the forward direction, which means that the behavior of eSBS-H (BF) is in this case somehow closer to plain IDA*. This explanation is confirmed by the results obtained when adopting a mixed policy, that we name BF/A, which still employs BF, but this time solves the tie-breaks using an alternate policy.

Algorithm	Nodes	States	Time
A*	2,615,014	2,615,014	24,152
SBS-A*	4,279,104	4,279,104	29,204
eSBS-A*	2,964,043	390,250	11,912
eSBS-A* *lite*	2,964,043	390,250	11,824
IDA*	17,388,654	–	20,554
SBS-IDA*	13,041,418	–	27,300
eSBS-H	12,945,026	1,275,697	10,613

Table 3: Results on the 50 Pancake puzzle data set. Time in ms. Results are averaged on 99 configurations.

Pancake puzzle

The N-pancake puzzle is a sorting problem, where the goal is to order the N integer numbers (i.e., pancakes of different sizes) and the allowed moves are permutations of the first $k = 2, \ldots, N$ (i.e., flipping the top k pancakes upside-down). Therefore, there is a constant branching factor of $N-2$ (if not considering the parent). We employed the GAP heuristic, recently shown to be the state-of-the-art within this domain (Helmert 2010), which can be easily extended to a front-to-front framework. The GAP heuristic iterates through the state and counts the number of consecutive pancakes who are not consecutive in their numbers. For each of these cases an unique operator must be activated on the optimal path. Thus, this heuristic is admissible.

We performed experiments with N ranging from 10 to 50, and using 100 random instances for each value of N: Table 3 shows the results for the 50-pancakes case on 99 cases that could be solved by all variants. We did not use a higher of number of pancakes (which was up to 60 in (Helmert 2010)), because that would have limited the number of configurations to compare with A*, which often runs out of memory with higher values of N. The jumping policy was the alternate policy (A). The results on the A* versions (top of the table) again show that all SBS versions generated more nodes than unidirectional A*. However, a large reduction of more than a factor of 2 was obtained in CPU time due to the caching and referencing techniques we used.

SBS-IDA* (bottom of the table) generated more nodes due to the lack of DD but its constant time per node is much smaller. eSBS-H generated fewer nodes and was the fastest algorithm among all variants. On the 50 puzzle, the average computational time of eSBS-H is half of that of plain IDA* with GAP heuristic (Helmert 2010).

It should be noted that the Pancake puzzle with GAP heuristic is not particularly favorable for our algorithm, since the heuristic is extremely well informed (the difference between the initial heuristic and the goal is 0.64 for N=50 using our 100 random puzzles); as it has been explained in the complexity analysis section, a scenario with perfect (or almost perfect) heuristic function is not the configuration where our framework can show its advantages at best.

Room Maps

A room map consists in a grid of rooms with random doors between them. Our experiments are performed on 5 different maps, obtained from the data set used in (Sturtevant et al. 2009; Felner et al. 2010), each having 32×32 rooms of size 7×7 and random paths that connect the rooms. In this domain IDA* would be inefficient due to many cycles and to the varying costs: for the same reason our hybrid algorithm is also inefficient in this domain. We concentrate on the variants that keep an OPEN-list.

Algorithm	JP	Nodes	States	Time	Win$_t$
Plain setting					
A*		8,592	8,592	70.0	–
SBS-A*	A	461,711	461,711	11,041.1	–
SBS-A*	BF	1,293,980	1,293,980	30,651.1	–
eSBS-A*	A	26,347	1,378	74.4	55 %
Random-weighted setting					
A*		8,831	8,831	70.9	–
SBS-A*	A	500,689	500,689	12,356.8	–
SBS-A*	BF	1,381,780	1,381,780	34,136.6	–
eSBS-A*	A	12,594	1,437	70.3	7 %
eSBS-A* *lite*	A	12,294	1,419	42.2	96 %

Table 4: Results on the room maps data set. JP stands for Jumping Policy. Top: plain setting with octile heuristic also adopted in (Felner et al. 2010). Bottom: setting with random-weighted edges. Time is measured in ms. Win$_t$ indicates the percentage of configurations where each algorithm is faster than A*.

We performed two kinds of experiments, always using the octile heuristic. The first setting is the one adopted by (Felner et al. 2010), where a cardinal move costs 1 and a diagonal move costs $\sqrt{2}$. In the second setting (“random-weight” room maps) we modified the map to better simulate real world scenarios where there are more than just two possible discrete values (1 or $\sqrt{2}$) for each move cost. In real scenarios the cost of each path depends on many details, such as road steepness, vehicle speed or the presence of obstacles, and it is therefore very unlikely to have exactly equal costs for many different edges. To simulate this, for each edge (i, j) we added a small random constant w_{ij} (where $w_{ij} \in [0, 10^{-2}]$, small enough not to reduce the information of the octile heuristic) to the cost of move $i \rightarrow j$. In both settings we used 500 random start/goal pairs (100 for each room map). Results are shown in Table 4.

The first setting presents a huge number of duplicate nodes mostly sharing the same g-value which makes the *lite* version of eSBS-A* not suitable, since, without using a CLOSED list, it cannot perform duplicate detection of nodes. Basic SBS-A* performs extremely worse than A* and eSBS-A*, visiting a huge number of nodes, which confirms the results obtained in (Felner et al. 2010). This happens because the room maps domain suffers of the tasks-vs-nodes blowup (see (Felner et al. 2010)), a condition which only marginally affects our eSBS-A* algorithm which uses the $best$-g-pruning technique. eSBS-A* overcomes this problem: it runs slightly faster than A* on 275 configurations out of 500 (55%), even if it is slower on average (70.0 vs. 74.9 ms).

In the second setting, again SBS-A* performs worse due the the blow-up problem. However, $best$-g pruning is much more effective in this setting (since there are less ties for

g-values) and eSBS-A* *lite* outperforms A*, running faster on 481 configurations (96%) and reducing average execution time by a factor of almost 2 (42.2 vs. 70.9 ms). In this setting the memory load of eSBS-A* gains scalability, and strong speed benefits derive from pruning and caching.

Moreover, it can be noticed that, in both settings, using the BF policy produces weak results, similarly to what happened in (Felner et al. 2010). This can be explained with the fact that, in the room maps, states with a low branching factor correspond to points close to walls in the map, and therefore they should not be preferred for expansion.

Summary and conclusions

Single-frontier Bidirectional Search (SBS) was recently introduced as an efficient front-to-front bidirectional search framework. In this paper we presented an enhanced version of the SBS search paradigm, which we called eSBS, by introducing the idea of state referencing within nodes, which heads to efficient techniques for pruning nodes and caching state successors. As a result, eSBS-A* overcomes many of the limitations of SBS-A* . The *lite* version has further potential to reduce the memory and the CPU time. Experimental results conducted over a range of different domains showed the great advantage of the eSBS-A* variants over the previous SBS-A*.

A second contribution of this paper was to present a new search algorithm, named eSBS-H, which is a hybrid between eSBS-A* and SBS-IDA*: when coupling the iterative deepening approach of IDA* with the eSBS framework, we obtained a very smart compromise between computational complexity and memory needs, which resulted to be an extremely efficient solution in all the tested domains. In particular, eSBS-H has a memory complexity which is the square root of A* and eSBS-A*

Future work will further investigate these algorithms with other jumping policies and will come up with more SBS variants.

Acknowledgments

We would like to thank Giovanni Soda and Carlos Linarez-Lopes for fruitful discussions and comments on an early version of this paper.

References

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

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N. R.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artif. Intell.* 175(9-10):1570–1603.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *SoCS 2010*. AAAI Press.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283–317.

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

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *AAAI*, 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.