Iterative-deepening Bidirectional Heuristic Search with Restricted Memory

Shahaf S. Shperberg[1], **Steven Danishevski**[2], **Ariel Felner**[2], **Nathan R. Sturtevant**[3]
[1] CS Department, Ben-Gurion University, Be'er-Sheva, Israel
[2] ISE Department, Ben-Gurion University, Be'er-Sheva, Israel
[3] Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
shperbsh@post.bgu.ac.il, stiven@post.bgu.ac.il, felner@bgu.ac.il, nathanst@ualberta.ca

Abstract

The field of bidirectional heuristic search has recently seen great advances. However, the subject of memory-restricted bidirectional search has not received recent attention. In this paper we introduce a general iterative deepening bidirectional heuristic search algorithm (IDBiHS) that searches simultaneously in both directions while controlling the meeting point of the search frontiers. First, we present the basic variant of IDBiHS, whose memory is linear in the search depth. We then add improvements that exploit consistency and front-to-front heuristics. Next, we move to the case where a fixed amount of memory is available to store nodes during the search and develop two variants of IDBiHS: (1) A*+IDBiHS, that starts with A*and moves to IDBiHS as soon as memory is exhausted. (2) A variant that stores partial forward frontiers until memory is exhausted and then tries to match each of them from the backward side. Finally, we experimentally compare the new algorithms to existing unidirectional and bidirectional ones. In many cases our new algorithms outperform previous ones in both node expansions and time.

1 Introduction and Overview

Search algorithms can be classified into three main categories with regards to the amount of memory they consume.

1. **Linear memory (LM)**. An algorithm may only store a single branch of the search tree (a path), and its memory consumption is $O(d)$ where d is the depth of the search.
2. **Fixed memory (FM)**. An algorithm is given a fixed amount of memory M (on top of the memory required for storing a single path) and must never exceed it. Cases 1 and 2 are denoted hereafter as **Restricted Memory (RM)**.
3. **Unrestricted memory (UM)**. Such algorithms are not restricted and usually use memory proportional to the size of the search tree that they explored. For example, A* (Hart, Nilsson, and Raphael 1968) stores all the nodes it generates in memory (either in OPEN or in CLOSED), which could grow polynomially or exponentially with the search depth. UM algorithms enable duplicate detection, which can potentially accelerate the search, but they cannot solve problems when memory is exhausted. Therefore, numerous RM algorithms have been developed for *unidirectional heuristic search* (UniHS).

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In *bidirectional heuristic search* (BiHS) the search is performed simultaneously from the $start$ and from the $goal$ until the two search frontiers meet. Significant advancements in BiHS were (recently) achieved in the UM category. A novel line of research was initiated with MM (Holte et al. 2016), a BiHS algorithm that *meets in the middle*, i.e. it will never expand a node whose g-value exceeds $C^*/2$ (where C^* is the optimal solution cost). Fractional MM (fMM) (Shaham et al. 2017) is a generalization of MM. Given a fraction p ($0 < p < 1$), fMM(p) is guaranteed not to expand nodes in the forward side whose g-value exceeds $p \cdot C^*$, and nodes in the backward side whose g-value exceeds $(1 - p) \cdot C^*$ (MM is fMM with $p = 1/2$). General Breadth-first Heuristic Search (GBFHS) (Barley et al. 2018) is a similar algorithm that sets the meeting points between the frontiers using a parameterized *split function* that limits which nodes are expanded from each side.

Nevertheless, unlike UniHS, the work on RM bidirectional search (RMBiS) is very limited (see Section 3.2). The aim of this paper is to start closing this gap. The main challenge for RMBiS is connecting the two frontiers, as they cannot both be fully stored without external memory (Sturtevant and Chen 2016). In an attempt to overcome this challenge, we propose several methods, all of which adapt the flexible meeting-point approach of fMM and GBFHS.[1]

We introduce a general iterative deepening bidirectional heuristic search algorithm (IDBiHS) that searches simultaneously from both directions while controlling the meeting point of the search frontiers. The basic variant of IDBiHS only stores a single path in memory (LM). It generalizes IDA* (Korf 1985) by running DFS from both search directions. For every forward frontier node n_F, a DFS is executed from the backward side in an attempt to find an optimal path to n_F. Next, we introduce two new fixed-memory BiHS algorithms (FM). Our first FM BiHS algorithm generalizes A*+IDA* (Bu and Korf 2019) by first running A* until M is exhausted. Then, it moves to IDBiHS, where the forward direction DFS starts from OPEN; this algorithm is therefore called A*+IDBiHS. Our second algorithm is called IDBiHS-Trans. It stores a partial set of forward fron-

[1] While there are other BiHS algorithms such as NBS (Chen et al. 2017) and DVCBS (Shperberg et al. 2019), they do not control the meeting point as needed in our approach.

tier nodes based on the memory available and tries to match the entire set with a backward DFS. We study variants and add improvements to all these algorithms.

Finally, we empirically evaluate all algorithms on several domains and compare them to existing ones. We show that IDBiHS outperforms IDA* in runtime and node expansions by up to a factor of 5.3. Furthermore, both A*+IDBiHS and IDBiHS-Trans improve upon IDBiHS and outperform existing FM methods by up to a factor of 3.

2 Definitions and Terminology

A shortest-path problem instance, I, is defined as a n-tuple $(G = (V, E), start, goal, h_F, h_B)$, where G is a graph, and $start, goal \in V$. In the shortest-path problem the aim is to find the least-cost path (with cost C^*) between $start$ and $goal$. UniHS algorithms search forward from $start$ to $goal$, while BiHS algorithms interleave two separate searches, a search forward from $start$ and a search backward from $goal$ until the search frontiers meet. $d(x, y)$ denotes the shortest distance between x and y, so $d(start, goal) = C^*$.

Front-to-end (F2E) algorithms use two heuristic functions, a forward heuristic h_F and a backward heuristic h_B, where for any node u, $h_F(u)$ estimates $d(u, goal)$, and $h_B(u)$ estimates $d(start, u)$. The *forward heuristic*, h_F, is *forward admissible* iff $h_F(u) \leq d(u, goal)$ for all u in V (of G) and is *forward consistent* iff $h_F(u) \leq d(u, u') + h_F(u')$ for all u and u' in G. The *backward heuristic*, h_B, is defined analogously. *Front-to-front* (F2F) BiHS algorithms use heuristics between any pair of states. In particular, $h(x, y)$ estimates $d(x, y)$ for any pair of states x, y. Finally, f_F, and g_F indicate f- and g-costs of nodes in the forward search, and f_B and g_B are similarly defined for the backward search.

3 Background

3.1 Restricted Memory UniHS Algorithms

IDA* (Korf 1985) is a benchmark LM UniHS algorithm. IDA* iterates on a threshold T, which is a lower-bound on C^* (initialized to be $f_F(start)$). At each iteration, IDA* performs a DFS from $start$ and prune nodes with $f > T$. The minimum f-value of the pruned nodes becomes the T value of the next iteration. The process repeats until $T = C^*$ and IDA* finds an optimal solution.

Many enhancements have been proposed for IDA* (Sarkar et al. 1991; Wah and Shang 1994; Burns and Ruml 2013; Stern et al. 2010; Bu et al. 2014; Sharon, Felner, and Sturtevant 2014; Hatem, Burns, and Ruml 2018), as well as other LM algorithms such as Dual IDA* (Zahavi et al. 2008), and IBEX (Helmert et al. 2019). These algorithms only store a single path and do not utilize the available memory.

To better utilize the available memory, many FM UniHS algorithms have been developed. A non-exhaustive list includes MREC (Sen and Bagchi 1989; Reinefeld and Marsland 1994), MA* (Chakrabarti et al. 1989), SMA* (Russell 1992), DBIDA* (Eckerle and Schuierer 1995), FPS (Schütt, Döbbelin, and Reinefeld 2013), breath-first heuristic search (Zhou and Hansen 2004) and A*+IDA* (Bu and Korf 2019).

3.2 Restricted Memory BiHS Algorithms

SFBDS. A notable LM BiHS algorithm is the IDA* variant of SFBDS (Felner et al. 2010; Lippi, Ernandes, and Felner 2016). A node in SFBDS is composed of two states, a forward state and a backward state, and a solution is found when both states are identical. When expanding a node either the forward state is forward expanded or the backward state is backward expanded. The decision of which state to expand is determined by a *jumping policy*. Given a fixed jumping policy, a tree is induced which can be searched using any admissible search algorithm. The IDA* variant of SFBDS applies IDA* on the induced tree to search for solutions. Since in SFBDS a search tree is induced by a jumping policy, when comparing to SFBDS, one should compare to specific existing jumping policies. The most promising jumping policy reported (Felner et al. 2010; Lippi, Ernandes, and Felner 2016) was the *jump if larger* policy (JIL(K)) that calculates the forward and backward heuristics after performing a lookahead to depth k. Then, JIL(k) chooses to expand the side with the larger h-value.

Perimeter Search Algorithms. Perimeter search (Dillenburg and Nelson 1994) is a class of FM BiHS algorithms. First, a perimeter around $start$ or around $goal$ is constructed. Then, a DFS is executed from the opposite direction until the perimeter is reached. Two variants of perimeter search are BIDA* and BAI which are covered next.

BIDA* (Manzini 1995), assumes a consistent F2F heuristic $h(u, v)$ between any two states and builds a perimeter P_d with a depth of d around the goal. BIDA* runs IDA* towards P_d and as a heuristic for a node n it uses $h_d(n) = \min_{m \in P_d}(h(n, m) + d(m, goal))$. Due to the consistency of the heuristic, BIDA* matches against a *relevant nodes* list within the perimeter. When a node n is explored by IDA*, nodes m within the perimeter for which $d(start, n) + h(n, m) + d(m, goal) > T$ are removed from the perimeter when exploring any of n's children. Therefore, BIDA* performs fewer heuristic evaluations as the depth of the search grows.

BAI (Kaindl et al. 1995) builds the perimeter by running reversed A* from the goal and searches towards the perimeter by a forward IDA*. BAI-Trans is a variant of BAI that allocates some memory to IDA* as well, to be used as transposition table, as done by Reinefeld and Marsland (1994). Finally, if the heuristic is consistent, BAI can be enhanced with the KKMax method from Kaindl and Kainz (1997). Let $fmin_B$ be the minimum f-value among all nodes in the perimeter. Nodes n with $g_F(n) - h_B(n) + fmin_B > T$ are pruned by IDA*. This enhancement results in Max-BAI and Max-BAI-Trans.[2]

Finally, Wilt and Ruml (2013) proposed a perimeter-based algorithm that dynamically maintains a perimeter from the backward side and runs IDA* from the forward side using the KKAdd method from Kaindl and Kainz (1997).

[2]Kaindl and Kainz (1997) also introduced Max-IDA*, which runs IDA* (without a perimeter) and switch the search direction every time T is increased, to apply the Max method. However, Max-IDA* was not significantly better than IDA*.

However, this algorithm increases the size of the perimeter during the search and is therefore not an RM algorithm.

3.3 The GBFHS Algorithm

General breadth-first heuristic search (GBFHS, Barley et al. (2018)) is a UM bidirectional heuristic search algorithm that iteratively increases the depth of the search. For each depth, denoted by $fLim$, GBFHS uses a *split function* (given as a parameter) that determines how deep to search on each side. The split function splits $fLim$ to $gLim_F$ and $gLim_B$, such that $fLim = gLim_F + gLim_B + \epsilon - 1$, where ϵ is the minimum edge cost (in unit edge cost domains $\epsilon - 1 = 0$). For a given iteration (i.e., a given value of $fLim$) all nodes with $f_D(n) \leq fLim$ and $g_D(n) < gLim_D$ are called *expandable*. GBFHS expands all the *expandable* nodes from both directions. GBFHS terminates if there exists a node n in both open lists with $g_F(n) + g_B(n) \leq fLim$. Otherwise, $fLim$ is incremented (a new iteration begins) and the split function updates either $gLim_F$ or $gLim_B$. The frontiers of GBFHS can be controlled to meet anywhere using a proper split function. fMM shares this property, as the two algorithms are often equivalent (Shperberg and Felner 2020). Our algorithms below adapt this principle of controlling the meeting points.

4 Linear Memory BiHS

We now introduce our new algorithm, iterative-deepening bidirectional heuristic search, or IDBiHS. We first present the LM variant and then proceed to FM variants.

IDBiHS uses thresholds similar to those of GBFHS, but IDBiHS is based on DFS iterations rather than using a best-first search mechanism through the use of open lists (as in GBFHS), thus it uses memory linear in the depth (LM).

The pseudo-code of IDBiHS is given in Algorithm 1. First, fT (the current iteration threshold, identical to $fLim$ in GBFHS) is initialized as $h(start, goal)$ (line 2). Then, an iterative process repeats until a solution is found, where each iteration corresponds to a new fT value (lines 4-8).

In each iteration the task is to find a solution of cost fT. If such a solution is not found, fT is incremented (using $nextT$ as described below), and a new iteration begins. Given fT, the meeting point of the current iteration is obtained by calling the *split function* which is given to the algorithm as a parameter. The split function determines the *meeting point* of the current iteration by setting a forward g-threshold (gT_F). The split function must be monotonically non-decreasing over successive iterations and must never return values greater than fT. Specific split functions that were used in our experiments are described in Section 7.

Algorithm 1: pseudo-code for IDBiHS

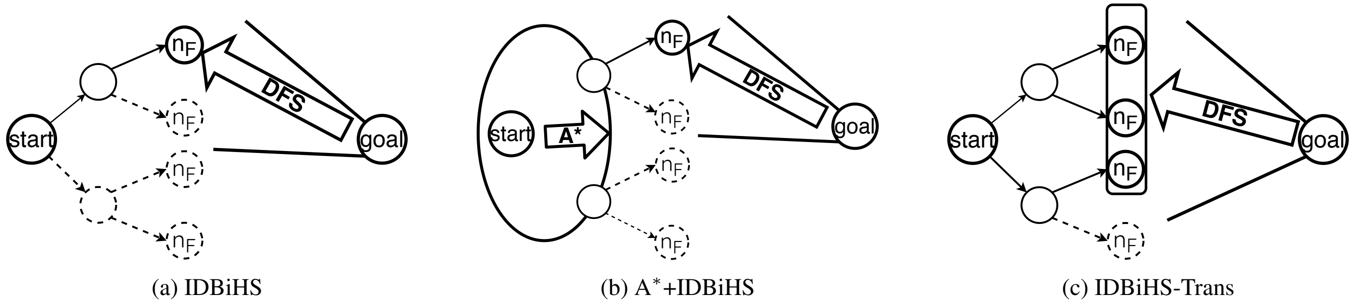
```
1  IDBiHS (s,g,h,split,ε)
2      fT, nextT ← h(start, goal)
3      path ← ∅
4      while true do
5          gT_F ← split(fT)
6          if F_DFS(s,g,fT,gT_F,h,path,nextT,ε)
7              return path
8          fT ← nextT
9  F_DFS (n_F,g,fT,gT_F,h,path,nextT,ε)
10     if f_F(n_F) > fT
11         updateNextBound(nextT,f_F(n_F),fT)
12         return false
13     if g_F(n_F) > gT_F
14         b_path ← ∅
15         gT_B ← fT − g_F(n_F) − ε
16         if B_DFS(n_F,g,fT,gT_B,h,b_path,nextT)
17             path ← path · b_path.reverse
18             return true
19         return false
20     foreach neighbour n of n_F do
21         path.push(state(n_F))
22         if F_DFS(n,g,fT,gT_F,h,path,nextT,ε)
23             return true
24         path.pop()
25 B_DFS (n_F,n_B,fT,gT_B,h,b_Path,nextT)
26     if state(n_B) = state(n_F) and g_F(n_F) + g_B(n_B) ≤ fT
27         return true
28     if f_B(n_B) > fT or g_B(n_B) > gT_B
29         updateNextBound(nextT,max(f_B(n_B), g_F(n_F) +
               g_B(n_B) + ε),fT)
30         return false
31     foreach neighbour n of n_B do
32         b_Path.push(state(n_B))
33         if B_DFS(n_F,n,fT,gT_B,h,bPath,nextB)
34             return true
35         else
36             b_Path.pop()
37 updateNextBound (nextT,f,fT)
38     if f > fT
39         nextT ← min(nextT, f)
```

4.1 Forward DFS

Once gT_F is obtained, a forward DFS procedure (F_DFS) is called from $start$ (lines 9-24). When F_DFS encounters a node n_F, it has three cases (line numbers in parentheses):

1. **Expand** (20-24). If $f_F(n_F) \leq fT$ and $g_F(n_F) \leq gT_F$, then expand n_F and move to one of its children.
2. **Prune** (10-12). If $f_F(n_F) > fT$, prune n_F and backtrack.
3. **Suspend and Match** (13-18). If $f_F(n_F) \leq fT$ and $g_F(n_F) > gT_F$, suspend F_DFS and call the backward DFS (B_DFS) in an attempt to match n_F from the backward side. As illustrated in Figure 1a, B_DFS is called for every forward frontier node n_F until either a solution is found or all forward frontier nodes have been explored.

When calling B_DFS on candidate nodes n_F, the g-threshold for the backward direction (gT_B) needs to be defined. Unlike GBFHS, where $gLim_B$ is known given $fLim$ and $gLim_F$ ($gLim_B = fLim - gLim_F - 1 + \epsilon$), in IDBiHS gT_B is defined specifically for each node n_F to be matched. For example, assume that $fT = 10$, $gT_F = 5$ and $\epsilon = 1$.



(a) IDBiHS (b) A*+IDBiHS (c) IDBiHS-Trans

Figure 1: Illustration of all IDBiHS variants

Further assume that F_DFS found a node n_F to be matched with $f_F(n_F) \leq 10$ and $g_F(n_F) = 7 > 5$. Then B_DFS needs only to expand nodes up depth 2 (gT_B= 2), thus generating nodes with cost 3 and matching n_F. However, if $g_F(n_F) = 6$, then nodes whose $g_B = 3$ also need to be expanded (gT_B= 3). This example shows that different gT_B values can be defined for the same fT and gT_F. Hence, we define: $gT_B(n_F) = fT - g_F(n_F) - \epsilon$ (line 15).

4.2 Backwards DFS

Upon reaching a candidate meeting node n_F in F_DFS and setting gT_B, B_DFS performs a DFS iteration from $goal$ (lines 25-36). When B_DFS encounters a node n_B, it has three options (line numbers in parentheses):

1. **Expand** (31-36). If $f_B(n_B) \leq fT$ and $g_B(n_B) \leq gT_B$, then expand n_B and move to one of its children.
2. **Prune** (28-30). If $f_B(n_B) > fT$, prune n_B and backtrack.
3. **Match** (26-27). If $f_B(n_B) \leq fT$ and $g_B(n_B) > gT_B$, match n_F against n_B. If they represent the same state, a solution has been found. Otherwise, n_B can be immediately pruned. [3]

After B_DFS finishes without matching n_F, B_DFS returns *false* and F_DFS resumes by backtracking from n_F. Note that in every iteration F_DFS is called once, while B_DFS is called many times, once for each frontier node.

As soon as all forward paths have been explored by F_DFS and no solution has been found, F_DFS returns *false* and a new iteration begins. Finally, the minimum f-value among all nodes pruned (from either direction) with $f > fT$ is a lower-bound on the cost of any solution that goes through them. Moreover, solutions that go through nodes n_B, that were pruned due to their g-value in an attempt to match a forward node n_F, are bounded by $g_F(n_F) + g_B(n_B) + \epsilon$. Therefore, next f-threshold is updated to be the minimum between all bounds (line 8).

4.3 Theoretical Analysis of IDBiHS

We now show that IDBiHS returns optimal solutions.

Lemma 1. *IDBiHS is guaranteed to return an optimal solution when given admissible heuristics.*

[3]When $\epsilon = 0$ nodes with $g_B(n_B) = gT_B$ should also be matched against n_F, but should not be pruned.

Proof. First, observe that IDBiHS cannot terminate before a solution of cost fT is found (while true loop at line 4) and that at every iteration of the while loop, fT is incremented. Assume by contradiction that IDBiHS did not find an optimal solution when $fT = C^*$. When $fT = C^*$, IDBiHS considers for expansion all nodes u, with $f_F(u) \leq C^*$ and $g_F(u) \leq gT_F$. Assume that a node u belongs to an optimal path $start, \dots, u, v, w, \dots, goal$, and that u is the last node in the path such that $d(start, u) \leq gT_F$. If the heuristic is admissible, the f-value of every node in the path cannot exceed C^*, therefore $f_F(u) \leq C^*$. Thus, the forward direction expands u and the backward search attempts to match v. Then, all nodes n' with $f_B(n') \leq C^*$ and $g_B(n') \leq C^* - g_F(v) - \epsilon$ are considered for expansion by the backward search. Since v is on an optimal path, $g_B(v) = C^* - g_F(v)$. In addition, $fT = C^*$. Since (v, w) is an edge in the search graph, we know that $d(v, w) \leq \epsilon$, and therefore $g_B(w) \leq g_B(v) - \epsilon$. Moreover, due to admissibility, $f_B(w) \leq C^*$, thus w fulfills both conditions, and is expanded by the backward search. Then, v is matched, and an optimal solution is found, a contradiction. □

In exponential unit edge cost domains where no heuristic is available, IDBiHS will generate $b^{(gT_F)}$ forward frontier nodes for each iteration (fT), where b is the branching factor. For every forward frontier node, B_DFS expands all nodes up to depth gT_B. Since we consider unit edge cost problems, $\epsilon = 1$ and $g_F(n_F) = gT_F + 1$ for every node n_F searched for by the backward search. Thus,

$$gT_B = fT - g_F(n_F) - \epsilon = fT - gT_F \tag{1}$$

This means that for every iteration, IDBiHS expands $b^{(gT_F)} \cdot b^{(gT_B)} = b^{(gT_F)} \cdot b^{(fT - gT_F)} = b^{(fT)}$ nodes, similar to IDA*. However, IDBiHS has worse performance in polynomial unit edge cost problems in which no heuristic is available. Assume that the number of expansions performed by a DFS up to depth d is bounded by d^C for some constant C. Therefore, IDA* will perform $(fT)^C$ expansions for each threshold fT. By contrast, IDBiHS expands $(gT_F)^C \cdot (fT - gT_F)^C$ nodes. For example, in a meet-in-the-middle policy where $gT_F = fT/2$, IDBiHS expands $(fT/2)^{2C}$ nodes, significantly more than the fT^C nodes expanded by IDA*. Nonetheless, we show that when a heuristic is available, IDBiHS can often outperform IDA* when applying the improvements described next.

H	F2F (Imp1) / Consistency (Imp2)			
	None	Imp1	Imp2	Both
G	10,767	**85**	3,728	**85**
G-1	8,808,663	6,571	110,171	**3,417**
G-2	597,465,611	218,698	5,891,237	**75,205**
G-3	7,699,462,563	8,272,225	139,889,666	**1,540,325**

Table 1: Node expansions with and without imp1 and imp2

5 Improving IDBiHS

The basic variant of IDBiHS can be further improved.

Improvement 1 (Imp1): If a front-to-front (F2F) heuristic is available, B_DFS can use it. In particular, given a forward frontier node n_F, when B_DFS reaches a backward node n_B the heuristic between n_F and n_B can be used. Thus, $f_B(n_B)$ in line 28 can be computed as $g_B(n_B) + h(n_B, n_F) + g_F(n_F)$.

Improvement 2 (Imp2): If the heuristic is known to be consistent then another improvement is possible. Kaindl and Kainz (1997) defined $Diff_F(n_F) = g_F(n_F) - h_B(n_F)$, and $Diff_B(n_B) = g_B(n_B) - h_F(n_B)$, corresponding to the *error* of $h_B(n_F)$ and $h_F(n_B)$ respectively. When trying to connect a node n_B generated by B_DFS to a given forward frontier node n_F, it holds that $f_B(n_B) + Diff_F(n_F) \leq d(n_B, start)$ and that $Diff_B(n_B) \leq d(n_B, goal)$. Therefore, the maximum of $f_B(n_B) + Diff_F(n_F)$ and $f_F(n_F) + Diff_B(n_B)$ can be used instead of $f_B(n_B)$ to improve the pruning (line 28). [4]

Since Imp1 and Imp2 are orthogonal, they can be combined if h is known to be both F2F and consistent.

5.1 Improvements Evaluation

We evaluated the effectiveness of these improvements over the original IDBiHS (given a meet-in-the-middle split policy) on the **Pancake Puzzle** with n pancakes (P[n]). We used the the GAP (G) heuristic (Helmert 2010). In addition, to get a range of heuristic strengths, we also used the G-k heuristics (for $k \in \{1, 2, 3\}$) where the k smallest pancakes are deleted from the GAP heuristic computation (Holte et al. 2016). For each combination of n and k, we used 100 random problems. The results are reported in Table 1, which presents the the average number of nodes expanded by IDBiHS for $n = 12$ pancakes. The results show that applying either Imp1 or Imp2 speeds up the search by up to a factor of 2,700 in terms of node expansions (See G-2 and Imp1) and by up to a factor 2,000 in terms of run-time compared to the basic variant. This difference becomes more significant as the heuristic deteriorates. However, combining both Imp1 and Imp2 has a diminishing effect. For GAP, it didn't contribute at all (85 for Imp1 and for both), while for GAP-3 using both improvements further reduced the number of expansions by a factor of 5.

In terms of runtime, IDBiHS is very efficient and has an overhead similar to IDA*. Imp1 runtime depends on the difficulty of obtaining a heuristic estimation between nodes. For example, in GAP-k, Manhattan distance (MD), and pattern databases (PDBs) (Culberson and Schaeffer 1998) on permutation problems such as Rubik's cube, there is no additional overhead for using F2F heuristic over F2E heuristic. However, using PDBs on the sliding-tile puzzle (STP) for F2F heuristics requires more memory and several more heuristic evaluations for each F2E evaluation due to the asymmetry introduced by the blank (Zahavi et al. 2008). For some problems (e.g., 4-peg Towers of Hanoi), it is not clear how to create efficient F2F heuristics and Imp1 is not applicable. Applying Imp2 requires two additional heuristic evaluations for each backward frontier node. When combining Imp1 and Imp2, the overhead from these extra evaluations is not significant, as most nodes are pruned by Imp1 and Imp2 is rarely applied. For example, the average runtime (seconds) in G-3 when using only Imp1 was 4.8 compared to 1.14 when using both improvements (x5.4 fewer node expansions and x4.2 less time).

The trends reported above are also evident when evaluating Imp1 and Imp2 on other domains (the domains used in in Section 7). Based on all this, it is certainly worthwhile to apply both improvements whenever possible. Therefore, only IDBiHS with both improvements is further evaluated and compared to other algorithms in Section 7.

6 Allowing Additional Memory

IDBiHS and its suggested improvements use memory linear in C^*. We now introduce two FM variants of IDBiHS.

6.1 A*+IDBiHS

The first algorithm, A*+IDBiHS (illustrated in Figure 1b), is a variation of IDBiHS inspired by A*+IDA* (Bu and Korf 2019). Given a memory budget M, A*+IDA* first runs A* until either a solution is found, or the memory used by A* exceeds M. Then, it continues by running IDA* starting from the OPEN nodes, denoted hereafter as N_F. Similarly, A*+IDBiHS executes A* from $start$ until it runs out of memory. Then, IDBiHS is executed sequentially from the nodes in N_F as follows. fT is initialized to be the minimal f-value in N_F. Next, F_DFS is executed on all nodes $n \in N_F$ for which $f_F(n) = fT$ in increasing order of h-values (the same ordering used by A*+IDA*). For each execution of F_DFS starting from a node $n \in N_F$, the next threshold variable ($nextT$) is initialized to be fT and is updated during F_DFS as explained above. Then, once F_DFS fails to find an optimal solution from n, $h(n)$ is updated to be the current $nextT - g(n)$.

Imp1 and Imp2 can also be applied to the IDBiHS searches in A*+IDBiHS. In addition, F_DFS can employ duplicate-detection (DD) and prune duplicate nodes that already appear in the closed list of A* or in N_F. However, DD induces additional overhead, as it requires computing hash functions for states, which can be expensive. Nonetheless, DD improved the performance of A*+IDBiHS and was therefore used in the experiments reported in Section 7.

Note that IDBiHS can also be bootstrapped using GBFHS instead of A*. The available memory is split between the two sides. But, doing this will weaken the strength of Imp1.

[4] Shaham et al. (2018) used similar ideas to define must-expand pairs for BiHS algorithms that assume heuristic consistency.

Assume that IDBiHS is bootstrapped by a BiHS, and that N_F, N_B are the forward and backward nodes in OPEN, respectively, after memory was exhausted. Now, F_DFS is executed from each node in N_F, and when a frontier node n_F is discovered, B_DFS must match n_F to every node in N_B. In practice, most nodes in N_B are pruned right away, since their F2F heuristic to n_F makes their f-value exceed fT. Thus, B_DFS iterations from such nodes are initiated and halt right away. In contrast, a B_DFS from $goal$ (e.g., IDBiHS and A*+IDBiHS) prunes common ancestors of these nodes without the need to visit them. While the unnecessary iterations over the nodes in N_B do not require additional node expansions, they consume a significant amount of time. In fact, we experimented with first running MM and then moving to IDBiHS when the memory budget is exhausted. Indeed, this algorithm required fewer node expansions than A*+IDA* and A*+IDBiHS, but it had a much larger runtime.

6.2 IDBiHS-Trans

We now introduce IDBiHS-Trans, a variant of IDBiHS that uses a transposition table to store nodes (illustrated in Figure 1c). While A*+IDA* stores nodes near $start$ to minimize the number of duplicate nodes on lower depths, IDBiHS-Trans stores frontier nodes in order to match against multiple nodes at once, and thus calling B_DFS fewer times (from the other frontier). In particular, if the available memory is sufficient to store K frontier nodes, then the number of B_DFS calls will be reduced by a factor of K. The pseudo-code of IDBiHS-Trans is very similar to basic IDBiHS (Algorithm 1), with a few small differences. Instead of calling B_DFS on line 16, in an attempt to match n_F (a forward frontier node), n_F is inserted into a transposition table. B_DFS is called only when the transposition table is full. Then, instead of matching against a single forward frontier node, B_DFS tries to match any of the nodes of the transposition table at once. If a solution was not found by B_DFS, the transposition table is discarded and F_DFS resumes.

Finally, Imp1 and Imp2 can be applied to IDBiHS-Trans. However, while Imp2 can be efficiently computed by using the minimal $\mathit{Diff}_F(k)$ value among all nodes $k \in K$, Imp1 is costly as F2F heuristic evaluation needs to be performed against each one of the K forward frontier nodes. Therefore, imp1 makes each B_DFS more expensive. Nonetheless, given a consistent heuristic, BIDA*'s method of saving heuristic evaluations can be applied (see Section 3). Specifically, when B_BFS considers a node n_B, every node $k \in K$ for which $g_F(k) + h(k, n_B) + g_B(n_B) > fT$ can be discarded when calling B_BFS on children of n_B.[5]

7 Empirical Evaluation

We performed experiments on three domains: **(1) The Pancake Puzzle** as described in Section 5.1. **(2)** The standard 100 instances of the **15 puzzle** (STP) problem (Korf 1985) using the MD heuristic. **(3) 8-Grid**-based pathfinding using the octile heuristic: 16 *brc* maps from Dragon Age Origins (DAO) with canonical ordering (Sturtevant 2012), each with 350 different start and goal points (a total of 5,600 instances). The edge cost of diagonal moves is 1.5. In all of these domains $\epsilon = 1$. Note that all of the above heuristic are consistent and can be used both as F2E and F2F.

All experiments were run on a machine with an AMD Ryzen 9 3900X CPU, on a single core and a 64GB RAM.

Our implementation is integrated into HOG2, a well-known, open-source, search platform. HOG2 implements common domains, heuristics and algorithms, and is used in numerous papers by many authors.[6]

7.1 Comparison of LM Algorithms

First, we compare the following LM algorithms: IDA*, SF-BDS with JIL(1) as a jumping policy,[7] and IDBiHS. IDBiHS was evaluated using two different split policies. The first policy, denoted as IDBiHS-0.5 splits each fT in the middle ($gT_F = (fT/2) - \epsilon$). The second policy was inspired by Pohls cardinallity criterion (Pohl 1971; Barley et al. 2018). The idea is to *balance the workload* (BW) between the two frontiers. IDBiHS-BW counts the number of nodes expanded in the previous iteration in the forward search, and those expanded in the backward search, and increments the g-threshold of the direction that expanded fewer nodes. The increment is the difference between the next iteration's fT and the current iteration's fT. In IDBiHS this means either increasing gT_F or leaving gT_F unchanged, which will automatically cause gT_B to increase, since fT is increased. We also experimented with a policy that considers the number of pruned nodes, but the results were similar to that of the BW split function.

Table 2 presents the averages number of node expansions before finding an optimal solution and the runtime for the LM algorithms. The task of finding an optimal solution is composed of two sub-tasks: finding a solution, and proving its optimality. In order to measure the effort invested in each of these two sub-tasks, we also report, in parenthesis next to the node expansions, what percentage of the overall expansions was performed in the last C-layer (i.e. after the algorithm proved that there is no solution of cost less than C^*). For reference, we also provide results in the "Best UM" column for the best among the results of A*, reverse-A* (A* from $goal$ to $start$) and MM. We used these algorithms because they are well-known and simple. In addition, they naturally bound the number of nodes expansions of our methods. A* bounds IDA* (and other algorithms that use IDA* iterations), reverse-A* bounds the effort of building the perimeter, and MM bounds IDBiHS with a meet-in-the-

[5]Alternatively, F2E search bounds (Alcázar, Riddle, and Barley 2020) such as KKMax, KKAdd (Kaindl and Kainz 1997), and the b search bounds (Sadhukhan 2012) can be used instead of F2F heuristic evaluations (Imp1) against every node in K. Naturally, these search bounds may be faster to compute than imp1 but they have a weaker pruning power.

[6]https://github.com/nathansttt/hog2. HOG2 is a general platform, thus, low-level and domain specific optimization tricks (.e.g., see (Burns et al. 2012)) that speedup the CPU overhead are not always implemented. However, the timing trends reported below will likely apply for more optimized and/or domain specific solvers.

[7]We also tried using a lookahead of 2, but the runtime overhead was too great and many problems timed-out without solution.

Domain	H	Expanded					Time(sec)				
		Best UM	IDA*	SFBDS JIL(1)	IDBiHS 0.5	IDBiHS BW	Best UM	IDA*	SFBDS JIL(1)	IDBiHS 0.5	IDBiHS BW
P[10]	G	21 (63%)	32 (72%)	72 (76%)	**31** (72%)	**31** (72%)	0.00	0.00	0.00	0.00	0.00
	G-1	371 (13%)	1,674 (58%)	2,068 (60%)	650 (57%)	**538** (60%)	0.00	0.00	0.00	0.00	0.00
	G-2	2,054 (9%)	43,571 (54%)	38,289 (56%)	10,178 (57%)	**9,845** (57%)	0.00	0.02	0.02	**0.01**	**0.01**
	G-3	3,369 (4%)	549,522 (53%)	511,224 (55%)	132,910 (56%)	**118,407** (55%)	0.01	0.25	0.28	**0.07**	**0.07**
P[12]	G	34 (57%)	95 (76%)	169 (79%)	85 (76%)	**83** (77%)	0.00	0.00	0.00	0.00	
	G-1	1,256 (14%)	9,665 (72%)	8,622 (74%)	3,417 (70%)	**2,857** (72%)	0.00	0.01	0.01	0.00	0.00
	G-2	13,089 (10%)	383,488 (68%)	205,469 (71%)	75,205 (70%)	**72,939** (72%)	0.04	0.25	0.15	0.06	**0.05**
	G-3	38,545 (6%)	8,601,396 (66%)	4,161,889 (69%)	1,540,325 (71%)	**1,390,806** (69%)	0.13	5.44	2.90	1.14	**1.03**
P[45]	G	45,822 (50%)	140,574 (79%)	184,809 (74%)	128,912 (78%)	**127,743** (78%)	2.32	0.94	1.25	0.85	**0.84**
STP	MD	8,143,628 (2%)	242,460,834 (50%)	139,225,772 (46%)	174,414,968 (40%)	**137,331,587** (48%)	24.76	47.85	35.90	36.44	**29.29**
Grid	Octile	383 (70%)	**47,060** (75%)	131,488 (78%)	210,353 (78%)	122,304 (79%)	0.00	**0.00**	0.01	0.01	0.01

Table 2: Average node expansions and runtime of linear-memory algorithms

middle policy.[8] Best UM should not be directly compared against the LM algorithms, but rather be used for estimating the performance loss from restricting the memory.

IDBiHS-BW is the best algorithm across all exponential domains. It outperforms IDA*, both in node expansions and time, by up to a factor of 5.3 and SFBDS by up to a factor of 4. The improvement is more significant for weaker heuristics. The runtime of IDBiHS-0.5 is slightly higher than that of IDBiHS-BW, but is still competitive, inferior only to SFBDS in STP. However, as the theoretical analysis suggests (Section 4.3) IDBiHS is outperformed by IDA* in the polynomial domain (Grid). Finally, the percentage of nodes expanded in the last C-layer is proportional to the heuristic strength. While for the UM algorithms most of the node expansions are usually performed before the last C-layer, for the LM algorithms most of the node expansions are performed in the last layer. For example, the entire last iteration of IDA* (from the root and onward) is performed after no solution was found with a smaller cost. Nonetheless, there is no significant different between the different LM algorithms in terms of the relative effort invested in the last C-layer.

7.2 Comparison of FM Algorithms

Table 3 presents results for FM algorithms. In order to consider meaningful amounts of memory, we used a memory budget proportional to the number of states stored by the Best UM algorithm (denoted by S). Specifically, we used $C \cdot S$, where $C \in \{50\%, 10\%, 1\%\}$. In most cases, IDBiHS-Trans requires the fewest expansions. However, it has a large runtime overhead per node. Nonetheless, IDBiHS-Trans achieved the fastest runtime in STP. Max-BAI performed well, achieving the fastest runtime and the least number of node expansions in GAP-1 through GAP-3, when having 50% memory. However, its performance deteriorated the most when less memory was available. By contrast, A*+IDBiHS uses less overhead per node, and therefore is often the fastest algorithm. We also compared to Max-BAI-Trans with different ratios between memory used for the perimeter and for the transposition table, but the results were similar to those of Max-BAI and are not reported.

All fixed memory algorithms experience a trade-off. Having more memory often results in fewer node expansions. However, more memory results in a larger overhead per node due to the cost of maintaining the required data-structures. For example, in STP, Max-BAI runs almost 4 times slower per node with 50% memory than with 10% memory. IDBiHS-Trans also demonstrates an anomaly, where using more memory result in more node expansions and the percentage of nodes expanded in the last C-layer is higher. The reason behind this anomaly is that forward frontier nodes are stored and matched by B_DFS only after the available memory is full. In the worst case scenario, an optimal solution can be found by matching only the first forward frontier node, but this node will only be matched once memory is filled by many other forward nodes. Thus, less memory ensures that fewer forward nodes are expanded. On the other hand, more memory causes fewer B_DFS executions, and thus fewer backward nodes are expanded. Therefore, the task of choosing how much memory to allocate is not trivial for any algorithm, especially for IDBiHS-Trans. Nonetheless, in most cases, it is more beneficial to use available memory than to not use memory at all (LM). In fact, IDBiHS-Trans has managed to run 2.7 times *faster* than the best UM algorithm (best among A*, reverse-A*, and MM) in STP. This improvement is partially due to the smaller overhead per node and partially due to less overall node expansions resulting from using consistency and the F2F heuristic.

8 Conclusions and Future Work

This paper presents IDBiHS, a general BiHS algorithm for memory-restricted algorithms. We started by introducing the basic variant which uses linear-memory. Then, we moved to fixed-memory algorithms and developed A*+IDBiHS, which bootstrap the search using A*, and IDBiHS-Trans, that uses a transposition table to store frontier nodes. An empirical evaluation suggests that our new methods often outperform existing methods in exponential domain, especially when given weaker heuristics. Nonetheless, all IDBiHS variants do not perform as well in polynomial domains. In addition, when the available memory is significantly large, both FM variants of IDBiHS can run slower and sometimes even expand more nodes than when using less memory. To over-

[8] Note that "Best UM" is not the best possible unrestricted memory algorithm, as there are other sophisticated algorithms which may be better than the ones we used (see for example Alcázar, Riddle, and Barley (2020)).

Domain	H	Mem	Expanded					Time(sec)				
			Max-BAI	BIDA*	A*+ IDA*	A*+ IDBiHS	IDBiHS-Trans	Max-BAI	BIDA*	A*+ IDA*	A*+ IDBiHS	IDBiHS-Trans
P[12]	G	50%	84(74%)	69(68%)	40(65%)	**36**(73%)	125(90%)	0.00	0.00	0.00	0.00	0.00
		10%	84(78%)	75(78%)	56(75%)	**48**(77%)	65(82%)	0.00	0.00	0.00	0.00	0.00
		1%	93(76%)	96(75%)	62(78%)	**54**(75%)	59(78%)	0.00	0.00	0.00	0.00	0.00
	G-1	50%	**858**(47%)	2,742(43%)	932(44%)	1,129(64%)	1,630(68%)	0.00	0.00	0.00	0.00	0.00
		10%	2,987(80%)	**950**(72%)	3,435(75%)	1,021(69%)	998(64%)	0.00	0.00	0.00	0.00	0.00
		1%	5,525(80%)	3,854(81%)	4,607(77%)	1,345(70%)	**1,203**(67%)	0.00	0.00	0.00	0.00	0.00
	G-2	50%	20,662(62%)	16,654(50%)	96,891(55%)	18,041(61%)	**8,243**(74%)	**0.03**	0.12	0.09	0.06	0.05
		10%	45,113(83%)	16,352(66%)	135,907(80%)	21,509(65%)	**6,670**(67%)	0.04	0.10	0.09	**0.03**	0.05
		1%	113,607(79%)	27,665(74%)	207,447(82%)	31,866(70%)	**11,924**(74%)	0.09	0.08	0.25	**0.03**	0.06
	G-3	50%	223,490(78%)	130,167(60%)	2,259,163(80%)	289,218(72%)	**35,915**(67%)	**0.26**	1.96	1.58	0.43	0.70
		10%	578,231(79%)	212,442(68%)	3,293,269(75%)	381,136(71%)	**41,020**(68%)	0.51	2.01	2.13	**0.36**	0.81
		1%	1,631,356(75%)	318,838(68%)	4,910,121(74%)	622,505(68%)	**104,516**(72%)	1.35	1.71	3.10	**0.50**	0.91
P[45]	G	50%	145,410(74%)	97,663(63%)	108,176(65%)	**87,563**(72%)	634,304(93%)	1.99	1.74	2.56	**1.61**	4.92
		10%	134,226(78%)	128,601(75%)	144,863(67%)	**120,340**(73%)	412,355(81%)	1.10	1.01	1.32	**0.87**	2.97
		1%	161,791(79%)	162,385(77%)	170,279(65%)	**143,120**(75%)	128,204(77%)	0.95	1.00	1.54	**0.84**	1.66
STP	MD	50%	7,998,667(46%)	7,654,126(40%)	38,347,881(65%)	15,489,868(42%)	**4,738,575**(50%)	20.83	29.16	27.94	24.02	**11.69**
		10%	13,804,939(64%)	5,183,115(51%)	71,470,823(74%)	24,322,900(41%)	**4,447,875**(47%)	9.50	13.17	17.04	12.22	**8.79**
		1%	55,391,222(66%)	6,026,913(56%)	118,002,945(68%)	46,075,139(39%)	**3,941,356**(35%)	24.83	13.18	23.41	13.59	**9.08**
Grid	Octile	50%	**8,362**(75%)	14,911(78%)	16,503(72%)	19,880(76%)	12,833(78%)	0.00	0.00	0.00	0.00	0.00
		10%	31,439(76%)	36,417(77%)	42,914(74%)	55,964(75%)	**14,388**(78%)	0.00	0.00	0.00	0.00	0.00
		1%	45,826(75%)	46,632(79%)	46,518(74%)	106,818(77%)	**40,545**(76%)	0.00	0.00	0.00	0.01	0.00

Table 3: Average node expansions and runtime of fixed-memory algorithms

come these limitations, we propose to continue the research in the following directions:

(1) Different hybrids of A*+IDBiHS and IDBiHS-Trans. While a large memory budget can deteriorate the performance of each of these algorithms individually, a combination of both algorithms might be better at utilizing the available memory and improve the perfromance. Instead of using all available memory either on A* or on a transposition table, we propose to use a portion of the memory on each.

(2) Storing Frontier Nodes in Bloom-filters. A Bloom filter (Bloom 1970) is a space-bounded data structure that stores a set of elements, and can answer a query of whether an element is a member of a set. We propose to modify IDBiHS-Trans to store *all* frontier nodes nodes in a Bloom-filter instead of storing some of them in a transposition table. Bloom-filters membership queries can return false-positive, but not false-negative. Therefore, when using Bloom-filters, states that are matched by B_DFS were not necessarily generated by the forward search. But, states that were not matched cannot possibly lead to a solution with a cost of fT. Thus, only nodes matched by the backward search need to be stored as potential meeting points, for further matching by the forward side. The hope is that the additional overhead induced by the false-positives will be lower than the one induced by the iterations on many forward frontier partial lists.

(3) Improving RMBiS in Polynomial Domains. IBEX is an LM algorithm (Helmert et al. 2019) that improves on IDA* in polynomial domains. The number of states that IDA* expands in polynomial domains is bounded by $O(n^2)$, where n is the number of states within the final threshold, while IBEX is bounded only by $O(n \log C^*)$. IBEX achieves this asymptotic improvement over IDA* by increasing the f-threshold in an aggressive way (based on ideas from exponential search and binary search). We believe that the same ideas can be also incorporated into RMBiS algorithms.

(4) Adapting more algorithms. Here we considered algorithms that control the meeting point of the two frontiers. It would be valuable to try to adapt other recent BiHS algorithms that do not control the meeting point (such as NBS (Chen et al. 2017), DVCBS (Shperberg et al. 2019), or BEA* (Alcázar, Riddle, and Barley 2020)) to the RM setting.

Acknowledgments

This work was supported by Israel Science Foundation (ISF) grant #844/17 to Ariel Felner and Eyal Shimony, by BSF grant #2017692, by NSF grant #1815660 and by the Frankel center for CS at BGU. This work was funded by the Canada CIFAR AI Chairs Program; we also acknowledge the support of NSERC.

References

Alcázar, V.; Riddle, P. J.; and Barley, M. 2020. A Unifying View on Individual Bounds and Heuristic Inaccuracies in Bidirectional Search. In *AAAI*, 2327–2334. AAAI Press.

Barley, M. W.; Riddle, P. J.; López, C. L.; Dobson, S.; and Pohl, I. 2018. GBFHS: A Generalized Breadth-First Heuristic Search Algorithm. In *SoCS*, 28–36. AAAI Press.

Bloom, B. H. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13(7): 422–426.

Bu, Z.; and Korf, R. E. 2019. A*+IDA*: A Simple Hybrid Search Algorithm. In *IJCAI*, 1206–1212. ijcai.org.

Bu, Z.; Stern, R.; Felner, A.; and Holte, R. C. 2014. A* with Lookahead Re-Evaluated. In *SoCS*, 44–52. AAAI Press.

Burns, E.; and Ruml, W. 2013. Iterative-deepening search with on-line tree size prediction. *Ann. Math. Artif. Intell.* 69(2): 183–205.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing Fast Heuristic Search Code. In *SoCS*, 25–32. AAAI Press.

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and Sarkar, S. C. D. 1989. Heuristic Search in Restricted Memory. *Artif. Intell.* 41(2): 197–221.

Chen, J.; Holte, R. C.; Zilles, S.; and Sturtevant, N. R. 2017. Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions. In *IJCAI*, 489–495. ijcai.org.

Culberson, J.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence* 14(3): 318–334.

Dillenburg, J. F.; and Nelson, P. C. 1994. Perimeter Search. *Artificial Intelligence* 65(1): 165–178.

Eckerle, J.; and Schuierer, S. 1995. Efficient Memory-Limited Graph Search. In *KI*, volume 981 of *Lecture Notes in Computer Science*, 101–112. Springer.

Felner, A.; Moldenhauer, C.; Sturtevant, N. R.; and Schaeffer, J. 2010. Single-Frontier Bidirectional Search. In *AAAI*, 59–64. AAAI Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4(2): 100–107.

Hatem, M.; Burns, E.; and Ruml, W. 2018. Solving Large Problems with Heuristic Search: General-Purpose Parallel External-Memory Search. *J. Artif. Intell. Res.* 62: 233–268.

Helmert, M. 2010. Landmark Heuristics for the Pancake Problem. In *SoCS*, 109–110. AAAI Press.

Helmert, M.; Lattimore, T.; Lelis, L. H. S.; Orseau, L.; and Sturtevant, N. R. 2019. Iterative Budgeted Exponential Search. In *IJCAI*, 1249–1257. ijcai.org.

Holte, R. C.; Felner, A.; Sharon, G.; and Sturtevant, N. R. 2016. Bidirectional Search That Is Guaranteed to Meet in the Middle. In *AAAI*, 3411–3417.

Kaindl, H.; and Kainz, G. 1997. Bidirectional Heuristic Search Reconsidered. *J. Artif. Intell. Res.* 7: 283–317.

Kaindl, H.; Kainz, G.; Leeb, A.; and Smetana, H. 1995. How to Use Limited Memory in Heuristic Search. In *IJCAI*, 236–242. Morgan Kaufmann.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.* 27(1): 97–109.

Lippi, M.; Ernandes, M.; and Felner, A. 2016. Optimally solving permutation sorting problems with efficient partial expansion bidirectional heuristic search. *AI Commun.* 29(4): 513–536.

Manzini, G. 1995. BIDA*: An Improved Perimeter Search Algorithm. *Artif. Intell.* 75(2): 347–360.

Pohl, I. 1971. Bi-directional search. In Meltzer, B.; and Michie, D., eds., *Machine Intelligence*, volume 6, 127–140. Edinburgh University Press.

Reinefeld, A.; and Marsland, T. A. 1994. Enhanced Iterative-Deepening Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 16(7): 701–710.

Russell, S. J. 1992. Efficient Memory-Bounded Search Methods. In *ECAI*, 1–5. John Wiley and Sons.

Sadhukhan, S. K. 2012. A new approach to bidirectional heuristic search using error functions. In *CSI*.

Sarkar, U. K.; Chakrabarti, P. P.; Ghose, S.; and Sarkar, S. C. D. 1991. Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds. *Artif. Intell.* 50(2): 207–221.

Schütt, T.; Döbbelin, R.; and Reinefeld, A. 2013. Forward Perimeter Search with Controlled Use of Memory. In *IJCAI*, 659–665. IJCAI/AAAI.

Sen, A.; and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, 297–302.

Shaham, E.; Felner, A.; Chen, J.; and Sturtevant, N. R. 2017. The Minimal Set of States that Must Be Expanded in a Front-to-End Bidirectional Search. In *SoCS*, 82–90.

Shaham, E.; Felner, A.; Sturtevant, N. R.; and Rosenschein, J. S. 2018. Minimizing Node Expansions in Bidirectional Search with Consistent Heuristics. In *SoCS*, 81–98. AAAI Press.

Sharon, G.; Felner, A.; and Sturtevant, N. R. 2014. Exponential Deepening A* for Real-Time Agent-Centered Search. In *AAAI*, 871–877. AAAI Press.

Shperberg, S. S.; and Felner, A. 2020. On the Differences and Similarities of fMM and GBFHS. In *SoCS*, 66–74. AAAI Press.

Shperberg, S. S.; Felner, A.; Sturtevant, N. R.; Shimony, S. E.; and Hayoun, A. 2019. Enriching Non-Parametric Bidirectional Search Algorithms. In *AAAI*, 2379–2386. AAAI Press.

Stern, R.; Kulberis, T.; Felner, A.; and Holte, R. 2010. Using Lookaheads with Optimal Best-First Search. In *AAAI*, 185–190. AAAI Press.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intell. AI Games* 4(2): 144–148.

Sturtevant, N. R.; and Chen, J. 2016. External Memory Bidirectional Search. In *IJCAI*, 676–682. IJCAI/AAAI Press.

Wah, B. W.; and Shang, Y. 1994. Comparison and Evaluation of a Class of IDA* Algorithms. *Int. J. Artif. Intell. Tools* 3(4): 493–524.

Wilt, C. M.; and Ruml, W. 2013. Robust Bidirectional Search via Heuristic Improvement. In *AAAI*. AAAI Press.

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.

Zhou, R.; and Hansen, E. A. 2004. Space-Efficient Memory-Based Heuristics. In *AAAI*, 677–682. AAAI Press.