ECAI 2024
U. Endriss et al. (Eds.)
© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA240991

On Parallel External-Memory Bidirectional Search

Lior Siag[a], Shahaf S. Shperberg[a,*], Ariel Felner[a] and Nathan R. Sturtevant[b,c]

[a]Ben-Gurion University of the Negev
[b]University of Alberta, Department of Computing Science
[c]Alberta Machine Intelligence Institute (Amii)

Abstract. Parallelization and External Memory (PEM) techniques have significantly enhanced the capabilities of search algorithms when solving large-scale problems. Previous research on PEM has primarily centered on unidirectional algorithms, with only one publication on bidirectional PEM that focuses on the meet-in-the-middle (MM) algorithm. Building upon this foundation, this paper presents a framework that integrates both uni- and bi-directional best-first search algorithms into this framework. We then develop a PEM variant of the state-of-the-art bidirectional heuristic search (BiHS) algorithm BAE* (PEM-BAE*). As previous work on BiHS did not focus on scaling problem sizes, this work enables us to evaluate bidirectional algorithms on hard problems. Empirical evaluation shows that PEM-BAE* outperforms the PEM variants of A* and the MM algorithm, as well as a parallel variant of IDA*. These findings mark a significant milestone, revealing that bidirectional search algorithms clearly outperform unidirectional search algorithms across several domains, even when equipped with state-of-the-art heuristics.

1 Introduction

A* [11] and its many variants are commonly used to optimally solve combinatorial and pathfinding problems. However, as these problems often involve state spaces of exponential size, practical limitations in terms of time and memory resources hinder the ability of these algorithms to tackle large instances, particularly when the scale of the problem increases (e.g., more variables, larger graphs, etc.). Thus, an important line of research in heuristic search is harnessing hardware capabilities to overcome these limitations. Parallelizing various components of the search process across multiple threads can lead to a significant reduction in running time. Similarly, exploiting external memory, such as large disks, allows algorithms to scale the size of OPEN and CLOSED lists, substantially increasing the size of problems that can be solved [6, 8, 9, 14, 23].

Orthogonally, recent research has focused on *bidirectional heuristic search* (BiHS) algorithms, which have been shown to outperform unidirectional search (UniHS) methods [1, 5, 31, 33], on some problems, but the majority of these results are on relatively small problems or with weakened heuristics. It has been conjectured [3] that bidirectional search does not perform well with strong heuristics, and it is unclear whether these results will scale to the largest problems. Thus, our aim is to scale bidirectional search algorithms to significantly larger problems and stronger heuristics.

The intersection of parallel and external memory (PEM) and BiHS has only been explored in the context of the meet-in-the-middle (MM) algorithm [13], yielding a variant called PEMM [35]. However, utilizing recent advancements in BiHS algorithms necessitates a framework for seamlessly converting both recent and future BiHS algorithms into corresponding PEM variants.

This paper builds upon previous research in parallel and external-memory search [35] to unify and explore how parallel computing and external memory utilization impact BiHS. In the interest of broad accessibility, we introduce a flexible framework capable of integrating any UniHS or BiHS algorithm into the PEM paradigm. Additionally, using this framework, we successfully integrate the state-of-the-art BiHS algorithm BAE* [27], which relies on a consistent heuristic, resulting in a variant called PEM-BAE*.

We empirically compare PEM-BAE* against PEM variants of A* (PEM-A*), reverse A* (PEM-rA*), and MM (PEMM) on 15-and 24-tile puzzles and random 20-disk 4-Peg Towers of Hanoi problems. In addition, we compare PEM-BAE* to a parallel variant of IDA*, called AIDA* [26], as well as its reversed version. Our results confirm that PEM-BAE* significantly outperforms the other algorithms as problem complexity increases. This superiority of PEM-BAE* is evident not only with weak heuristics, as previously demonstrated for BAE*, but also persists when utilizing strong heuristics on large problems. This highlights that a BiHS algorithm stands as the state-of-the-art approach for addressing multiple challenging problems.

2 Background and Definitions

Research on bidirectional heuristic search (BiHS) has spanned several decades, dating back to the work of Pohl [25]. Recently, a new theoretical understanding emerged regarding the necessary nodes for expansion during the search [7, 29], sparking a series of algorithms that find optimal [1, 4, 5, 31], near-optimal [2], and memory-efficient solutions [32]. Other research has delved into algorithm comparisons [1, 30, 33] and explored the potential advantages of BiHS over Unidirectional search (UniHS) [37].

The focus of this paper is on two algorithms: MM, the first algorithm to be integrated with parallel external-memory search, and BAE*, chosen due to its superior performance. BAE* and other BiHS algorithms have demonstrated strong performance, surpassing their UniHS counterparts, on small problems with relatively weak heuristics. Some have suggested that, as the scale of problems increase, BiHS algorithms can maintain their dominance over UniHS algorithms, even when stronger heuristics are employed [1, 34, 37]. Others have conjectured that bidirectional search will not perform well on problems with strong heuristics [3]. This paper evaluates these conjectures by constructing a PEM variant of BAE* capable of solv-

* Email: shperbsh@bgu.ac.il

ing large problems with state-of-the-art heuristics.

2.1 BiHS: Definitions and Algorithms

In BiHS, the aim is to find a least-cost path, of cost C*, between $start$ and $goal$ in a given graph G. $dist(x, y)$ denotes the shortest distance between x and y, so $dist(start, goal) = C^*$. BiHS executes a forward search (F) from $start$ and a backward search (B) from $goal$ until the two searches meet. BiHS algorithms typically maintain two open lists OPEN_F and OPEN_B for the forward and backward searches, respectively. Each node is associated with a g-value, an h-value, and an f-value (g_F, h_F, f_F and g_B, h_B, f_B for the forward and backward searches). Given a direction D (either F or B), we use f_D, g_D and h_D to indicate f-, g-, and h-values in direction D.

The g-value of a state s is cost of the best path discovered to s, and f-value of a state is the sum of its g- and h- values. Most BiHS algorithms consider the two *front-to-end* heuristic functions [15] $h_F(s)$ and $h_B(s)$ which respectively estimate $dist(s, goal)$ and $dist(start, s)$ for all $s \in G$. h_F is *forward admissible* iff $h_F(s) \leq dist(s, goal)$ for all s in G and is *forward consistent* iff $h_F(s) \leq dist(s, s') + h_F(s')$ for all s and s' in G. Backward *admissibility* and *consistency* are defined analogously.

BiHS algorithms mainly differ in their node- and direction-selection strategies and other termination criteria. We next describe MM and BAE*, both implemented in this paper.

Meet in the middle (MM). MM [13] is a BiHS algorithm ensuring that the search frontiers *meet in the middle*. In MM, nodes n in OPEN_D are prioritized by:

$$pr_D(n) = \max(f_D(n), 2g_D(n)) \tag{1}$$

MM expands the node with minimal priority, $PrMin$ on both OPEN_F and OPEN_B. MM halts the search once the following two conditions are met: (1) the same node n is found on both OPEN lists; (2) the cost of the path from $start$ to $goal$ through n is $\leq LB_{\text{MM}}$ where LB_{MM} is a lower bound on C^* that is computed as follows:

$$LB_{\text{MM}} = \max(PrMin, fMin_F, fMin_B, gMin_F + gMin_B) \tag{2}$$

where $fMin_F$ ($fMin_B$) and $gMin_F$ ($gMin_B$), are the minimal f- and g-values in OPEN_F (OPEN_B), respectively. Sturtevant and Chen [35] provided a PEM variant of MM (PEMM). Our framework below is an extension of PEMM.

BAE*. Most algorithms (e.g., MM) only assume that the heuristics used are admissible. BAE* [27, 1] (and the identical algorithm DIBBS [28]) are BiHS algorithms that specifically assume that both h_F and h_B are consistent and thus exploit this fact. Let $d_F(n) = g_F(n) - h_B(n)$, the *difference* between the actual forward cost n (from $start$) and its heuristic estimation to $start$. This indicates the *heuristic error* for node n (as $h_F(n)$ is a possibly inaccurate estimation of $g_B(n)$). Likewise, $d_B(m) = g_B(m) - h_F(m)$. BAE* orders nodes in OPEN_F according to

$$b_F(n) = f_F(n) + d_F(n) \tag{3}$$

$b_F(n)$ adds the heuristic error $d_F(n)$ to $f_F(n)$ to indicate that the opposite search using $h_B(n)$ will underestimate by $d_F(n)$. Likewise, $b_B(m) = f_B(m) + d_B(m)$ is used to order nodes in OPEN_B. At every expansion cycle, BAE* chooses a search direction D and expands a node with minimal b_D-value. Additionally, BAE* terminates once the same state n is found on both OPEN lists and the cost of the path from $start$ to $goal$ through n is $\leq LB_B$ where LB_B, is a the following lower bound on C^* (known as the b-bound):

$$LB_B = (bMin_F + bMin_B)/2 \tag{4}$$

where $bMin_D$ is the minimal b-value in OPEN_D.

Given a consistent heuristic, BAE* was proven to return an optimal solution. $b(n)$ is more informed than other priority functions (as it also considers $d(n)$), and was shown to outperform common unidirectional and bidirectional algorithms [1, 33] on relatively simple domains. These included the 15-puzzle with the Manhattan Distance heuristic, the 12-disk 4-peg Towers of Hanoi Problem, grid benchmark problems, and the pancake puzzle with a weakened GAP heuristic. While BAE* stands as a state-of-the-art BiHS algorithm, it has not been tested on large sliding-tile or Towers of Hanoi problems, or with large pattern database heuristics. In this paper, we develop PEM-BAE*, a PEM version of BAE*, which is able to scale to far larger problems and heuristics.

2.2 Parallel External-Memory Search

External Memory Search structures a search such that the maximum size of a problem solved scales according to the size of the disk, instead of available RAM [24]. These algorithms are often paired with parallel search methods, as techniques to minimize random I/O often group states together, allowing parallel processing.

Two classes of external memory search appear in the literature. One class aims to perform a complete breadth-first search of a state space, employed for verifying state space properties [18] or building large heuristics [14]. These approaches maintain information about every state on disk, loading portions of the data into memory for expansion and duplicate detection. Another class of algorithms, including External A* [9] and search with structured duplicate detection [40, 41], is used to solve large problem instances. In these algorithms, OPEN is stored explicitly on disk, and CLOSED may or may not be stored, depending on the properties of the state space [22].

Both classes aim to reduce I/O operations to disk, e.g., by delaying operations like duplicate detection until many states can be processed in parallel [17], and dividing the state space up into smaller buckets of states [9, 12, 19, 36, 38] which are stored together on disk, and then loaded and expanded together. When duplicate states all hash into the same bucket, it reduces the complexity of checking for duplicates. Hash-based duplicate detection [18] and sorting-based duplicate detection [20] take buckets of states where duplicate detection has been delayed, load them into memory, and remove duplicates using hash tables or sorting, respectively. Structured duplicate detection [40] structures the state space so that all successor buckets can be loaded into memory for immediate duplicate detection.

External memory search has often relied on exponential-growing state-spaces with unit costs to ensure sufficient states are available to efficiently process in parallel. Notably, algorithms like PEDAL [12] have extended these approaches to non-unit-cost problems.

3 The PEM-BiHS Framework

We next introduce a high-level framework called *Parallel External Memory Bidirectional Heuristic Search* (PEM-BiHS), into which both BiHS and UniHS can be seamlessly integrated. PEM-BiHS is designed to efficiently solve very large problem instances. Leveraging parallelization capabilities along with using external memory, PEM-BiHS utilizes the foundations laid by algorithms such as

Algorithm 1 PEM-BiHS General Framework

```
 1: procedure PEM-BiHS ($start$, $goal$)
 2:     $U \leftarrow \infty, LB \leftarrow$ ComputeLowerBound()
 3:     $\textsc{Open}_F$, $\textsc{Open}_B$, $\textsc{Closed}_F$,$\textsc{Closed}_B \leftarrow \emptyset$
 4:     Push($start$, $\textsc{Open}_F$)                ▷ create bucket and record
 5:     Push($goal$, $\textsc{Open}_B$)
 6:     while $\textsc{Open}_F \neq \emptyset \wedge \textsc{Open}_B \neq \emptyset \wedge U > LB$ do
 7:         $D \leftarrow$ ChooseDirection()
 8:         $b \leftarrow$ ChooseNextBucket($\textsc{Open}_D$)
 9:         ParallelReadBucket($b$, $D$)      ▷ including In-Bucket DD
10:         RemoveDuplicates($b$, $\textsc{Closed}_D$)
11:         CheckForSolution($U$, $b$, $\textsc{Closed}_{\bar{D}}$)
12:         ParallelExpandBucket($b$, $\textsc{Open}_D$)
13:         WriteToClosed($b$, $\textsc{Closed}_D$)
14:         $LB \leftarrow$ ComputeLowerBound()
15:     return $U$
```

PEMM, PEDAL, External A*, and structured duplicate detection and exploits further parallelization opportunities.

We begin with a high-level description of PEM-BiHS together with the pseudo-code presented in Algorithm 1, which builds on [35], followed by a detailed description of algorithmic components. PEM-BiHS initializes an OPEN and CLOSED list for each direction (line 3). These lists do not explicitly store search nodes; instead, they maintain references to files (buckets) that contain the corresponding nodes. PEM-BiHS employs the following stages:

Halting condition (line 6): During each expansion cycle, PEM-BiHS evaluates the cost U of the current incumbent solution in comparison to the calculated lower bound LB, derived from the nodes within the open lists. If $U \leq LB$ or one of the open lists is empty, PEM-BiHS halts and returns the current solution cost (or infinity if no solution was found). Otherwise, the search continues.

Choosing direction and bucket for expansion (line 7–8): next, PEM-BiHS chooses the search direction D (UniHS always chooses the forward side) as well as a bucket from $\textsc{Open}_D$ to be expanded.

Retrieving the bucket (line 9): Next PEM-BiHS performs parallel reading of the file containing the bucket from external memory into the internal memory (RAM). This stage involves eliminating duplicate states within the bucket.

Duplicate Detection (DD) (line 10): PEM-BiHS then eliminates duplicates nodes with other buckets in $\textsc{Closed}_D$.

Solution Detection (line 11): PEM-BiHS checks whether a new solution was found.

Expansion (line 12 detailed in Algorithm 2): Nodes from memory are concurrently expanded, generating children. These children are then written to their respective buckets (creating new ones if needed, and their references are inserted into $\textsc{Open}_D$).

Writing to disk (line 13): Finally, the expanded nodes are written to disk, creating a new duplicate-free bucket. A reference to this bucket is inserted into CLOSED.

We next turn to provide an in-depth description of different algorithmic components of PEM-BiHS.

3.1 State-space Representation

Typically, states are represented using high-level structures for convenient programming. In the context of combinatorial puzzles (e.g., the 15- and 24-puzzles), states are commonly stored as an array, with each cell's value corresponding to the label of the object it holds. To pack these states into files, an additional encoding/decoding mechanism is required for converting states into bits and decoding them. These decodings aim to minimize memory consumption and reduce I/O time. For example, in the 24-puzzle practical implementations often use 8 bits (byte) to represent the identity of each tile demanding 200 bits (8×25). Leveraging bit manipulation allows compression to 125 bits (5 bits are enough to store the identity of a tile). Furthermore, recognizing that a state is a permutation of 0-24 enables to use only $\lceil \log_2(25!) \rceil = 84$ bits, e.g., using the Lehmer encoding, which maps each permutation to a unique integer in the range $\{1 \dots 25!\}$.

3.2 Bucket Structure

Nodes are grouped into buckets based on pre-determined attributes or *identifiers*. E.g., in the context of External A*, a bucket groups nodes with identical g- and h-values (e.g., all nodes n with $g(n) = 3$ and $h(n) = 4$ belong to the 3-4-bucket). In PEMM, a bucket groups nodes with identical priority ($pr(n)$, as defined for MM) as well as identical g-value. Alternatively, a hash value of a state, obtained by applying a hashing function to divide the set of states into a fixed number of values, can also serve as a property for defining a bucket.

Importantly, PEM-BiHS assumes that an entire bucket can fit into memory (in addition to a fixed-sized cache used to store successors before flushing them to disk, as detailed in Section 3.7). Therefore, bucket identifiers may not allow too many nodes to be mapped into a single bucket, although adaptive methods have been used to dynamically adapt bucket sizes [39]. Buckets are then written to files and loaded into memory as needed.

In PEM-BiHS, OPEN and CLOSED are maintained inside main memory. They store bucket *records*, which include the bucket identifiers and a link to the file containing the bucket. During an expansion cycle, a bucket record from OPEN is chosen for expansion. That bucket's file is then loaded into memory. Different algorithms within PEM-BiHS will choose different buckets as described next.

3.3 Direction, Prioritization, and Lower-bound

Selecting direction (line 7 in Algorithm 1). Numerous strategies can be employed for direction selection in BiHS. Three prevalent strategies appear in the BiHS literature: i) choosing the direction with minimal priority (as employed by MM), ii) alternating between search directions, and iii) Pohl's cardinality criterion, which chooses the direction with the smaller open-list. Naturally, UniHS algorithms consistently choose the same direction.

Prioritizing buckets (line 8). Selecting the next node to expand is the essence of any search algorithm. PEM-BiHS allows the use of any priority function, under the restriction that the priority function must induce a total order among buckets based on the bucket identifiers. Thus, a bucket is chosen by comparing bucket identifiers within OPEN. For example, in the implementation of a PEM variant of A* using this framework (denoted as PEM-A*), the bucket identifier includes the g- and h-values of nodes.

It is well known that the priority function influences the number of nodes expanded. For example, A* prioritizes nodes based on lower f-values, and often adds a second prioritization criterion ("tie-breaking" between nodes that share the same f-value), which prefers nodes with higher g-value (and thus lower h-value). This "higher-g-first" tie-breaking typically reduces the number of expanded nodes compared to "lower-g-first", particularly in domains with unit edge costs. Nevertheless, in standard A*, different tie-breaking policies usually expand states at the same rate (nodes per second). By contrast, in the context of PEM-BiHS, tie-breaking policies can also significantly affect the node expansion rate. If PEM-A* breaks ties according to lower-g-first, once a g-h-bucket is expanded, new nodes will never be added to it, due to the monotonically increasing nature of g-values. By contrast, when using a higher-g-first policy, nodes can be added to the bucket after its expansion. Consequently, the same bucket can be selected for expansion multiple times, up to a number that scales quadratically with the total number of buckets, instead of only once. Thus, while using the lower-g-first tie-breaking may result in more expansions when compared to higher-g-first, the gains from minimizing the costly I/O operations by the lower-g-first tie-breaking make it a more resource-effective strategy. We have validated this in an empirical study on the 15-puzzle problem with a PDB heuristic (see details in Section 5). PEM-A* with the lower-g-first policy expanded 4.4 times more nodes than the higher-g-first policy (616,758 vs. 2,724,974), but still was able to run four times faster (7.8 vs 32 seconds).

Computing Lower-bound (line 14). Different lower-bounds may be employed by PEM-BiHS algorithms [33]. For instance, A* uses the minimal f-value in the open list as a lower bound on the solution cost, MM uses LB_{MM} (Eq. 2) and BAE* uses LB_B (Eq. 4). These and other bounds can be plugged into our framework.

3.4 Reading Buckets

In prior PEM search studies, the process of reading a bucket (file) was carried out sequentially, influenced by the constraints of Hard Disk Drives (HDDs). Concurrent threads accessing the same file on HDDs could lead to performance degradation. However, the recent, popular Solid-State Drives (SSDs) not only provide faster memory access but also benefit from parallelized reading. Therefore, PEM-BiHS further optimizes performance by parallelizing the reading process. To parallelize the reading of a bucket, PEM-BiHS distributes the file containing the bucket equally among multiple threads. In our experiments, this resulted in a twofold speed-up compared to sequential reading. Subsequently, each thread reads states from the disk and decodes them into their in-memory state representation (See Section 3.1).

3.5 In-bucket Duplicate Detection

To minimize I/O operations, the elimination of duplicates is delayed until a bucket is chosen for expansion. Duplicate nodes may arise within the same bucket when a state is discovered via different paths. Additionally, duplicates can occur within closed buckets if the state has already been expanded with a lower g-value, or within other open buckets if states have been discovered with the same g-value. Notably, if the bucket identifiers include the g-value, other open buckets cannot contain duplicates of the same state with the same g-value, and can be disregarded for duplicate detection. During the reading phase, we manage in-bucket duplicate detection. However, to minimize I/O, we postpone duplicate detection within the closed list (as discussed in Section 3.6) until after all nodes have been read.

Similar to hash-based delayed duplicate detection (DDD) [17, 18], when loading a bucket into memory, its nodes are placed into a hash table based on their states using a perfect hash function, and duplicates are ignored if their cell is already filled. Unlike the work of Korf [18], PEM-BiHS allows multiple threads to read from the same bucket concurrently. To support this, each unique hash value is paired with a mutex. Once the reading and in-bucket duplicate detection is done, all threads are synchronized.

Algorithm 2 Parallel Bucket expansion pseudo-code

```
 1: procedure ParallelExpandBucket($b$, $D$)
 2:     cache ← $\emptyset$
 3:     for every state $s$ in $b$ do
 4:         if current thread should expand $s$ then
 5:             for each successor $s_i$ of $s$ do
 6:                 $sb$ ← GetBucketOfNode($s_i$)
 7:                 if $sb$ not in Open_D then
 8:                     AddBucket(Open_D, $sb$)
 9:                 AddState(cache_$sb$, $s_i$)
10:                 if cache_$sb$ is full then
11:                     FlushToDisk(cache_$sb$)        ▷ Also locks $sb$
12:     if cache is not empty then
13:         FlushToDisk(cache)
```

3.6 Duplicate Detection against CLOSED

To identify and eliminate duplicates of in-memory nodes with respect to nodes stored in closed buckets (line 10), a scanning process is initiated which reads relevant buckets (in small increments) and compares them against the in-memory nodes. With *Delayed Duplicate Detection* [17], each bucket is associated with a hash value and exclusively contains states assigned that particular value by a hash function. As a result, the scanning process is confined to closed buckets sharing the same hash value as the state of the node under consideration. For instance, if a node's state has a hash value of 5, only buckets associated with the hash value 5 are relevant and considered for duplicate detection.

In PEM-BiHS, the determination of the relevant buckets for scanning relies on the identifiers that define them. If the bucket records include hash values of states, a similar approach to Korf [17] can be employed. Alternatively, if the bucket record contains the h_D-value of states, for direction D, as an identifier, only buckets with the same h_D-value need to be scanned. Therefore, if a bucket record contains both the h_F-value and the h_B-value, the scanning process is limited to buckets that possess identical values for both h_F and h_B.

In addition, with unit edge-cost undirected graphs, (where edges can be followed in both ways) there are three cases for finding duplicates of a node generated at level x. (1) A parent p at level $x-2$ generates a child node n at level $x-1$. The child node n generates its parent again at level x. (2) Consider a cycle of even length k which was first explored by the search at the ancestor node a at level 0. The farthest node of this cycle will be seen twice at level $x = k/2$ from two parents which are at level $x-1$. (3) Consider a cycle with odd length k. Here, the two nodes farthest from a will be generated at level $y = \lfloor k/2 \rfloor$ and each will generate the other at level $x = y + 1$. Note that duplicates only occur as a result of a cycle, and a cycle can be either even or odd. Thus, when generating node n with $g(n) = x$ we only need to check buckets with g-values of $x-2$, x, and $x-1$, for these three cases, respectively. Consequently, using the g-value as an identifier could significantly reduce the number of buckets that need to be scanned. This approach optimizes the duplicate detection process based on the available information in the bucket records.

3.7 Parallel Node Expansion

The process of parallel bucket expansion (line 12) is outlined in Algorithm 2. To enable parallel expansion, each thread is allocated an equal portion of the nodes. Newly generated nodes are inserted into a dedicated successor cache for each thread. Each cache is divided into smaller arrays, where each array corresponds to a specific bucket record. This avoids the necessity of immediately writing each new successor to the file, thereby minimizing I/O operations.[1] The framework supports reopenings by adding a new bucket with the same identifier to OPEN and merging it with the corresponding CLOSED bucket after its expansion. In our evaluation, we used consistent heuristics, so we did not encounter any reopenings. Once an array reaches its capacity, all its nodes are flushed into the disk, generating new buckets on disks as well as new bucket records in OPEN if necessary. To prevent simultaneous writes to the same file and maintain data integrity, each bucket record is linked to a mutex, which is locked when a thread writes to a bucket. This design maintains data integrity and allows concurrent writing.

Note that bucket expansion and delayed solution detection in this framework (see Section 3.8) can occur concurrently with separate threads, as delayed solution detection is only concerned with the nodes that are about to be expanded, which are already loaded into memory and will not change due to the expansion.

3.8 Solution Detection

There are two approaches for solution detection (Algorithm 1, line 11): *immediate solution detection* (ISD) and *delayed solution detection* (DSD). In ISD, solutions are identified upon generation of a node n. This involves a query to the open list of the opposite frontier to check for the existence of a node with the same state as the newly generated node n. This is trivially implemented in UniHS, as it only entails checking if the state of n is the goal. Similarly, in standard BiHS, when the open lists are stored in memory, ISD can be efficiently performed using a hash table or a direct-access table where each item can be accessed in constant time.

However, when buckets are stored on disk, ISD can lead to frequent I/O calls every time a node is generated or when the cache of generated nodes is filled. Therefore, PEM-BiHS employs DSD as suggested by Sturtevant and Chen [35]. In DSD, solutions are identified during the expansion phase once all states of the bucket that was chosen to be expanded are already loaded into memory and stored in a hash table. Following that, closed buckets from the opposite direction are loaded in segments and compared against the hash table of expanded nodes in an effort to identify a solution. It's important to highlight that only relevant buckets need to be loaded; for instance, if the identifiers of the bucket of expanded nodes include h_F and h_B values, only closed buckets corresponding to the same h-values need to be considered. We note again that when the expansion and solution detection are done, all threads are synchronized.

It is important to acknowledge that the benefits of DSD come with certain trade-offs. First, since solutions are detected at a later stage, some nodes might be expanded which could have been avoided with ISD. Furthermore, search bounds leveraging information across the minimal edge cost (often denoted as ϵ) cannot be employed with DSD. This limitation arises because these bounds rely on ISD to improve the lower bound (LB) on the solution cost (we refer the reader to Sturtevant and Chen [35] for more details).

4 Parallel External-Memory BAE*

We next describe the implementation details needed for obtaining a PEM variant of the BAE* algorithm (PEM-BAE*).

Direction, Prioritization, and Lower-bound. The prioritization of buckets relies on the BAE* priority function (Eq.3). The lower bound is determined by the b-bound (Eq.4). Additionally, the direction selection policy alternates between directions; a seemingly simplistic approach that has proven to be as effective as more intricate policies [1]. In PEM, this means that the two search sides take turns in loading buckets into memory and expanding them.

Bucket Structure. In PEM-BAE*, we classify nodes into buckets based on their g_D-, h_F-, and h_B-values. This bucket structure offers several advantages. First, it encapsulates all the essential information needed for computing the BAE* priority function. Consequently, during each expansion cycle, only a single bucket needs to be loaded into memory, enhancing efficiency. Second, the g-value, serving as one of the bucket identifiers, can sometimes further reduce the number of closed buckets scanned during duplicate detection (as discussed in Section 3.6). Finally, as the classification is based on three values, the resulting buckets tend to remain relatively small. Notably, when buckets become too large for memory, a hash value can be introduced as an additional identifier but such a scenario did not arise in our experiments. A possible limitation of this approach is the potential for significant imbalances in the distribution of nodes across buckets. This bucket size imbalance could impede runtime performance, as we explore in our experiments below.

Solution Detection. PEM-BAE* uses DSD.

5 Experimental Results

We performed experiments with PEM-BiHS on the 15- and 24-sliding-tile puzzles (STP) and 4-peg Towers of Hanoi (ToH4). All experiments were executed on 2 Intel Xeon Gold 6248R Processor 24-Core 3.0GHz, 192 GB of 3200MHz DDR4 RAM, and 100TB SSD for the external memory. By default, all parallel algorithms were assessed utilizing all 96 available virtual threads (with 48 physical cores). Nevertheless, we have also conducted an ablation study to investigate algorithm performance while varying the number of threads.

We tested the following algorithms besides PEM-BAE*. First, we instantiated both A* and MM within the PEM-BiHS framework, resulting in PEM-A* and PEMM, respectively. Similarly to A*, PEM-A* consistently expands nodes in the forward direction, employs the minimal f-value in OPEN as a lower bound for the solution cost (LB), and expands a g-h bucket with the minimal f-value during each expansion cycle. PEM-A* uses the low-g-first tie-breaking as detailed in Section 3.3. PEM-A* incorporates ISD, checking for solution upon node generation, where the PEM-BAE* and PEMM employ DSD, checking for solutions before node expansions. As BiHS search algorithms explore from both the $goal$ and the $start$ states, it is crucial to ensure that any potential advantage is not merely a consequence of asymmetries, which causes the search tree from one side to be much smaller. Such asymmetries could also be leveraged in a unidirectional search from $goal$ to $start$. To address this concern, we executed a reverse variant of PEM-A*, denoted as PEM-rA*, where the search is conducted from $goal$ to $start$.

PEMM uses the direction-selection, lower-bound, and prioritizations of MM while using the lower-g-first tie-breaking. PEMM uses the g-value and priority value (Eq. 1) as bucket identifiers. As a BiHS, PEMM adopts DSD as part of its operational strategy.

For comparison, we have also implemented Asynchronous Parallel IDA* (AIDA*), [26]). AIDA* is a parallelized adaptation of IDA* which conducts a breadth-first search to a predetermined depth. The resulting frontier is subsequently distributed among all threads. Additionally, we've evaluated the reverse version of AIDA*, referred to as rAIDA*. Lastly, we evaluated the standard versions of A* and

[1] One may perform a DD inside this cache array. This will have a marginal effect on the overall performance.

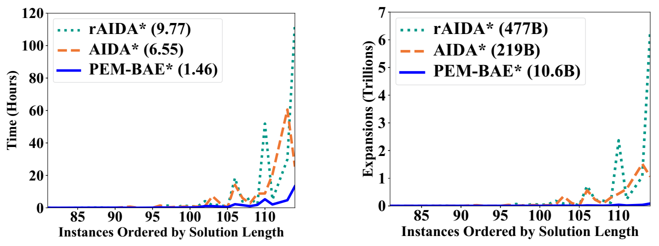

Figure 1. 24-puzzle results: (left) runtime, (right) node expansions

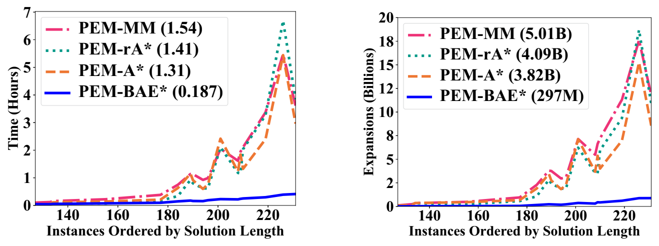

Figure 2. ToH4 16+4 results: (left) runtime, (right) node expansions.

	MD		PDB	
	Time	Expansions	Time	Expansions
All instances				
AIDA*	3.45	451,421,959	0.43	7,762,927
rAIDA*	**2.44**	335,167,556	**0.37**	6,118,084
PEM-A*	102.33	56,542,721	2.01	2,724,974
PEM-rA*	84.38	43,451,519	1.85	2,302,668
PEM-MM	16.49	26,771,047	5.2	2,572,780
PEM-BAE*	6.11	**3,113,271**	3.06	**626,440**
A*	56.88	15,549,689	2.22	615,155
BAE*	**10.13**	**2,707,414**	**1.76**	**453,988**
IDA*	51.82	242,460,834	2.67	3,456,177
The 10 hard instances: 3, 15, 17, 32, 49, 56, 60, 66, 82, 88				
AIDA*	22.18	2,943,505,999	2.13	46,314,389
rAIDA*	16.67	2,695,821,070	**1.93**	41,047,358
PEM-A*	901.19	350,840,875	7.8	17,124,704
PEM-rA*	786.58	308,829,220	6.67	14,371,919
PEM-MM	74.14	165,459,580	13.07	14,989,610
PEM-BAE*	**13.31**	**15,749,202**	6.05	**3,199,891**
A*	378.41	98,596,826	13.63	3,636,711
BAE*	**56.69**	**13,865,491**	**10.33**	**2,451,979**
IDA*	368.81	1,731,811,022	16.79	22,069,583

Table 1. 15-puzzle Results. Avg. time in seconds.

BAE*, without parallelization or external memory.

5.1 15-Puzzle

We first experimented on Korf's 100 random instances of the 15-puzzle [16]. This domain is relatively compact (13^{10} states) and could be solved without external memory. However, its size enables a comprehensive comparison of all algorithms across different heuristics before moving to larger domains. For heuristics, we used Manhattan Distance (labeled MD) and a 3-4-4-4 additive pattern database [10]) (labeled PDB). To construct the PDB, we divided the puzzle into four squares, one for each corner, with each pattern also including the blank. The average runtime (in seconds) and the number of node expansions are presented in Table 1 (top).

Naturally, using the PDB heuristic substantially reduced both the time and node expansions for all algorithms when compared to using the MD heuristic. The AIDA* variants expanded the largest number of nodes but exhibited the fastest overall runtime. Due to their DFS nature, the AIDA* variants do not store nodes in memory, let alone external memory, nor do they involve sorting nodes, as typically done by best-first search algorithms. Thus, they have the smallest time overhead per node in comparison with all other algorithms. Among the PEM algorithms, both BiHS algorithms outperformed the UniHS algorithms in terms of node expansions and runtime, when using the MD heuristic. Notably, PEM-BAE* significantly outperformed *all* PEM or iterative deepening algorithms in terms of node expansions by an order of magnitude. This finding is consistent with prior studies that compared BAE* to A* [1, 33], underscoring the advantages of harnessing the consistency of heuristics, as demonstrated by the performance of both A* and BAE* as presented in the table. Even when employing the PDB heuristic, PEM-BAE* maintained a significant edge over all other PEM or iterative deepening algorithms in terms of node expansions. However, it exhibited slower runtime than PEM-A* and PEM-rA*, a phenomenon we address in Section 5.5. Furthermore, despite its parallelism, PEM-BAE* was slower than both A* and BAE* when using the PDB heuristic. As anticipated, in scenarios where the problems are relatively simple to solve, the overhead incurred by utilizing external memory and parallelization serves to decelerate the search process rather than expedite it.

Table 1 (bottom) provides results on the ten most difficult instances (with largest solution cost). In these instances, the superiority of PEM-BAE* is more pronounced in terms of node expansions, and it also outperforms PEM-A* and PEM-rA* in terms of runtime. Additionally, PEM-BAE* outperformed both AIDA* and rAIDA* with MD. Although PEM-BAE* was still slower than AIDA* and rAIDA* with PDB heuristic, the performance gap is comparatively narrower.

As the problems become more challenging or, conversely, when the heuristic weakens, the performance of PEM-BAE* improves relative to other algorithms. This is demonstrated next on the 24-Puzzle, marking the first evaluation of BiHS algorithms for this large domain.

5.2 24-Puzzle

We experimented with the 50 24-puzzle problems of Korf and Felner [21], using a 6+6+6+6 additive PDB heuristic coupled with its reflection about the main diagonal [10]. Given the immense size of these 24-puzzle problems and the extensive computation time they demand, assessing all algorithms becomes impractical. Moreover, the memory demands for tackling these problems are substantial, making it impractical to execute standard (in-memory) A* and BAE* algorithms. Thus, we compared PEM-BiHS with the AIDA* variants. Figure 1 illustrates the runtime (left) and the number of expanded nodes (right) for each instance. Instances are sorted solution length in ascending order, which roughly indicates the problem's difficulty. For instances with the same solution length, we averaged the results. The plot legends also display the average runtime and expansions for each algorithm across all instances, shown in parentheses.

In general, PEM-BAE* performs the best in both node expansions and runtime. On average, PEM-BAE* expands only 4.4% of the nodes expanded by AIDA* and runs 4.5 times faster. These findings align with the observed trend in the 15-puzzle, indicating that on challenging problems, PEM-BAE* outperforms UniHS algorithms even when equipped with state-of-the-art (or near state-of-the-art) heuristics. The average disk space consumption of PEM-BAE* for the 5x5 STP instances was 653GB, with a peak of 4TB. This underscores the impracticality of employing in-memory algorithms (such as A* and BAE*) for tackling these challenging problems.

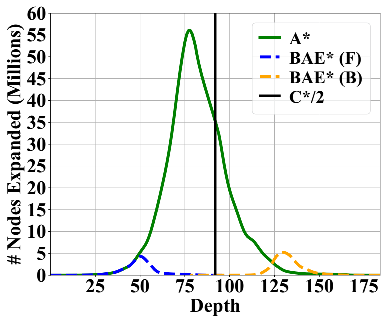

Figure 3. Distribution of node expansions in ToH4 instance with solution length of 184

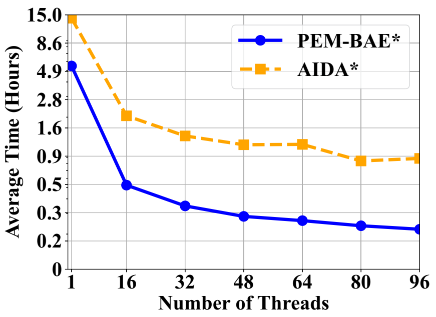

Figure 4. Avg. runtime as a function of number of threads in 24-Puzzle

	PEM-BAE*	PEMM	PEM-A*	PEM-rA*
15-STP$_{\text{MD}}$	509,537	**1,623,472**	552,553	514,950
15-STP$_{\text{PDB}}$	204,719	494,765	**1,355,708**	1,244,685
15-STP$^H_{\text{MD}}$	1,183,261	**2,231,718**	389,308	392,623
15-STP$^H_{\text{PDB}}$	528,908	1,146,871	**2,195,475**	2,154,710
TOH4	439,122	**908,970**	811,984	808,777
Average	573,109	**1,281,159**	1,061,006	1,023,149

Figure 5. Avg. nodes expanded per second; ^{H} indicates the hard instance in STP.

5.3 4-peg Towers of Hanoi

In TOH4, we examined 20 random start and goal pairs with 20 disks, utilizing a 16+4 additive PDB heuristic [10]. In this domain, numerous cycles exist, posing a challenge for algorithms that lack duplicate detection, as already noted by Felner et al. [10]. This issue is so severe that neither AIDA* nor rAIDA* could solve a single problem even after running for days. Consequently, we only compared PEM-BAE*, PEM-A*, PEM-rA*, and PEM-MM.

The results, presented in Figure 2, highlight a significant performance gap between PEM-BAE* and the other algorithms. On average, PEM-BAE* runs 7 times faster than its UniHS counterparts and expands a factor of 12.9 fewer nodes. Notably, PEMM was approximately 1.17 times slower than both PEM-A* and PEM-rA*, and it expanded more nodes than both of them.

5.4 Analyzing Previous Conjectures

In a previous analysis of bidirectional search [3], it was suggested that in a unidirectional search "if the majority [*of nodes*] are expanded at shallower depth than the solution midpoint [$C^*/2$] then [...] a bidirectional heuristic search would expand more nodes than a unidirectional heuristic search." . The analysis for this claim predates our current understanding of BiHS. But, since we have the data it is worthwhile to evaluate the validity of this claim.

In this context, in our experiments we found that in 19 of 20 Towers of Hanoi instances and all 15-puzzle instances when using the PDB, the unidirectional algorithm (PEM-A*) expanded the majority of states prior to the solution midpoint. Yet, in all of these problems PEM-BAE* expanded fewer node than PEM-A*. Thus, the previous analysis does not hold for PEM-BAE* on these problems. This analysis for a representative ToH4 instance is presented in Figure 3. It is a matter for future work to analyze these claims in more depth and to consider whether or how to revise them to be more accurate.

5.5 Analyzing Expansion-per-second Ratios

The Table in Figure 5 presents the number of nodes expanded per second (NPS) by different PEM algorithms on the domains in which all PEM algorithms were evaluated, namely 15-STP and TOH4. While these algorithms differ in how they choose their search direction, decide on which bucket to expand, and terminate the search, these differences are not expected to significantly impact NPS. Two factors, however, have the potential to affect NPS. First, BiHS algorithms perform DSD, while UniHS algorithms perform ISD. DSD generally requires more computational effort, though a profiling analysis revealed that this additional time was negligible. Second, differences in bucket structure can affect the number of buckets and their sizes, affecting the I/O time of the algorithms. The results show significant variations in NPS among the algorithms. Despite its relative strength in nodes and in time, PEM-BAE* exhibited the worst (smallest) NPS overall, suggesting that an alternative bucket structure might further improve its advantages over the other algorithms.

5.6 Ablation Study on the Number of Threads

To assess the thread utilization of PEM-BAE*, we compare its performance against AIDA* on the 24-STP while varying number of threads. Specifically, we conducted experiments using 1, 16, 32, 48, 64, 80, and 96 (virtual) threads on a subset of problems (problems 4, 36, 45, 48) with an intermediate solution cost, $C^* = 100$.

Figure 4 shows average runtimes for various thread configurations, with a logarithmic y-axis. As observed previously, PEM-BAE* surpasses AIDA* performance with 96 threads and consistently outperforms it across varying thread counts. As anticipated, adding more threads yields diminishing returns. PEM-BAE* reduced its runtime by a factor of 11 when transitioning from 1 thread to 16 threads, whereas AIDA* improved by a factor of 7. Beyond 16 threads, runtime reduction becomes smaller, decreasing only by a factor of 2 for both algorithms when transitioning from 16 to 96 threads. This reduced gain can be attributed to memory access required for obtaining heuristic values, imbalanced subtrees and last-layer expansions for AIDA*, constrained I/O parallelization (relative to the thread count), imbalanced bucket sizes, and locking overhead for PEM-BiHS.

6 Conclusions and Future Work

We presented PEM-BiHS, a parallel external-memory (PEM) BiHS framework for single-target search in undirected, uniformly weighted search graphs, which was used to create a PEM variant of BAE* (PEM-BAE*). Our empirical evaluations show that PEM-BAE* outperforms UniHS algorithms both in runtime and node expansions, even with well-informed heuristics. These findings challenge the conjecture put forth by Barker and Korf [3], suggesting that BiHS algorithms would not significantly surpass UniHS or bidirectional brute-force search. Further theoretical study is necessary to analyze our results in relation to this conjecture, as well as to other theoretical comparisons between BiHS and UniHS algorithms [13, 37].

Future research could also address NPS differences among algorithms by exploring dynamic bucket sizes and other approaches that relax the best-first assumption, aiming to achieve more balanced buckets [12].

Acknowledgements

This work was supported by the Israel Science Foundation (ISF) grant #909/23 awarded to Shahaf Shperberg and Ariel Felner, by Israel's Ministry of Innovation, Science and Technology (MOST) grant #1001706842, awarded to Shahaf Shperberg, and by United States-Israel Binational Science Foundation (BSF) grant #2021643 awarded to Ariel Felner. This work was also partially funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the National Sciences and Engineering Research Council of Canada (NSERC).

References

[1] V. Alcázar, P. J. Riddle, and M. Barley. A unifying view on individual bounds and heuristic inaccuracies in bidirectional search. In *The Thirty-Fourth Conference on Artificial Intelligence, AAAI 2020*, pages 2327–2334, 2020.

[2] D. Atzmon, S. S. Shperberg, N. Sabah, A. Felner, and N. R. Sturtevant. W-restrained bidirectional bounded-suboptimal heuristic search. In *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, pages 26–30, 2023.

[3] J. K. Barker and R. E. Korf. Limitations of front-to-end bidirectional heuristic search. In *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence, AAAI*, pages 1086–1092, 2015.

[4] M. W. Barley, P. J. Riddle, C. L. López, S. Dobson, and I. Pohl. GBFHS: A generalized breadth-first heuristic search algorithm. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS*, pages 28–36, 2018.

[5] J. Chen, R. C. Holte, S. Zilles, and N. R. Sturtevant. Front-to-end bidirectional heuristic search with near-optimal node expansions. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*, pages 489–495, 2017.

[6] R. Döbbelin, T. Schütt, and A. Reinefeld. Building large compressed PDBs for the sliding tile puzzle. In *Computer Games: Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3, 2013, Revised Selected Papers 2*, pages 16–27. Springer, 2014.

[7] J. Eckerle, J. Chen, N. R. Sturtevant, S. Zilles, and R. C. Holte. Sufficient conditions for node expansion in bidirectional heuristic search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS*, pages 79–87, 2017.

[8] S. Edelkamp and S. Schrödl. *Heuristic Search - Theory and Applications*, chapter Distributed Search. Academic Press, 2012.

[9] S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *Annual Conference on Artificial Intelligence*, pages 226–240. Springer, 2004.

[10] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *J. Artif. Intell. Res.*, 22:279–318, 2004.

[11] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] M. Hatem, E. Burns, and W. Ruml. Solving large problems with heuristic search: General-purpose parallel external-memory search. *Journal of Artificial Intelligence Research*, 62:233–268, 2018.

[13] R. C. Holte, A. Felner, G. Sharon, N. R. Sturtevant, and J. Chen. MM: A bidirectional search algorithm that is guaranteed to meet in the middle. *Artif. Intell.*, 252:232–266, 2017.

[14] S. Hu and N. R. Sturtevant. Direction-optimizing breadth-first search with external memory storage. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1258–1264, 2019. URL https://webdocs.cs.ualberta.ca/~nathanst/papers/DEBFS.pdf.

[15] H. Kaindl and G. Kainz. Bidirectional heuristic search reconsidered. *J. Artif. Intell. Res.*, 7:283–317, 1997.

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

[17] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI*, pages 650–657, 2004.

[18] R. E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6):1–40, 2008.

[19] R. E. Korf. Minimizing disk I/O in two-bit breadth-first search. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI*, pages 317–324, 2008.

[20] R. E. Korf. Comparing search algorithms using sorting and hashing on disk and in memory. In *IJCAI*, pages 610–616, 2016.

[21] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artif. Intell.*, 134(1-2):9–22, 2002.

[22] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.

[23] D. Kunkle and G. Cooperman. Twenty-six moves suffice for rubik's cube. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 235–242, 2007.

[24] D. Kunkle and G. Cooperman. Solving rubik's cube: disk is the new RAM. *Communications of the ACM*, 51(4):31–33, 2008.

[25] I. Pohl. Bi-directional search. *Machine intelligence*, 6:127–140, 1971.

[26] A. Reinefeld and V. Schnecke. Work-load balancing in highly parallel depth-first search. In *Proceedings of IEEE scalable high performance computing conference*, pages 773–780. IEEE, 1994.

[27] S. K. Sadhukhan. Bidirectional heuristic search based on error estimate. *CSI Journal of Computing*, 2(1-2):S1, 2013.

[28] E. C. Sewell and S. H. Jacobson. Dynamically improved bounds bidirectional search. *Artif. Intell.*, 291:103405, 2021.

[29] E. Shaham, A. Felner, J. Chen, and N. R. Sturtevant. The minimal set of states that must be expanded in a front-to-end bidirectional search. In *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS*, pages 82–90, 2017.

[30] S. S. Shperberg and A. Felner. On the differences and similarities of fMM and GBFHS. In *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS*, pages 66–74, 2020.

[31] S. S. Shperberg, A. Felner, N. R. Sturtevant, S. E. Shimony, and A. Hayoun. Enriching non-parametric bidirectional search algorithms. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI*, pages 2379–2386, 2019.

[32] S. S. Shperberg, S. Danishevski, A. Felner, and N. R. Sturtevant. Iterative-deepening bidirectional heuristic search with restricted memory. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS*, pages 331–339, 2021.

[33] L. Siag, S. Shperberg, A. Felner, and N. Sturtevant. Front-to-end bidirectional heuristic search with consistent heuristics: enumerating and evaluating algorithms and bounds. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI*, pages 5631–5638, 2023.

[34] L. Siag, S. Shperberg, A. Felner, and N. R. Sturtevant. Comparing front-to-front and front-to-end heuristics in bidirectional search. In *Proceedings of the International Symposium on Combinatorial Search, SoCS*, volume 16, pages 158–162, 2023.

[35] N. R. Sturtevant and J. Chen. External memory bidirectional search. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI*, pages 676–682, 2016.

[36] N. R. Sturtevant and M. J. Rutherford. Minimizing writes in parallel external memory search. In *IJCAI*, pages 666–673. IJCAI/AAAI, 2013.

[37] N. R. Sturtevant, S. S. Shperberg, A. Felner, and J. Chen. Predicting the effectiveness of bidirectional heuristic search. In *ICAPS*, pages 281–290. AAAI Press, 2020.

[38] Á. Torralba, V. Alcázar, P. Kissmann, and S. Edelkamp. Efficient symbolic search for cost-optimal planning. *Artif. Intell.*, 242:52–79, 2017.

[39] R. Zhou and E. Hansen. Dynamic state-space partitioning in external-memory graph search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, pages 290–297, 2011.

[40] R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI*, pages 683–689, 2004.

[41] R. Zhou and E. A. Hansen. Parallel structured duplicate detection. In *AAAI*, pages 1217–1224, 2007.