BiBFIDS: Memory-Efficient Bidirectional Bloom Filter-Based Search

Idan Yankelev, Yarin Yerushalmi Levi
{idanyan, yarinye} @ post.bgu.ac.il

Abstract

Bidirectional search includes searching from both the start and goal nodes simultaneously for more efficient pathfinding. Most existing methods prioritize the time and number of expansions required to find the optimal path rather than the memory required for the search process, which is crucial under restricting resource limits. In this project, we investigate utilizing Bloom filters to reduce an algorithm's memory usage and to find optimal paths under restrictions on memory usage during the search. Our experiments on common game problems show that our method finds the optimal solution in a comparable time with minimal memory usage.

1 Introduction

In this project, we introduce a novel bidirectional search approach that leverages Bloom filters' (BF) efficiency to enhance search algorithms' performance in solving complex combinatorial problems. Compared to unidirectional search, bidirectional search simultaneously explores the search space from both the start and the goal states, significantly reducing the number of nodes that need to be expanded. However, one of the primary challenges in bidirectional search is the efficient management of the search frontiers from each side. This is particularly challenging in large, intricate search spaces with redundant node expansions that lead to increased memory usage. Our proposed method addresses these challenges by integrating BF, a probabilistic data structure known for its space efficiency in handling large sets, into the bidirectional search framework. By using BF to compare the frontiers from both sides, we efficiently detect overlaps between the two searches, significantly reducing the memory footprint and minimizing unnecessary node expansions. This innovation not only improves the overall efficiency of the search process, but also makes bidirectional search more feasible for large-scale and memory-limited environments. We evaluate our approach using a variety of combinatorial puzzles, including the Pancake Problem, the Sliding Tile puzzle, and the Top-Spin puzzle, comparing it against established search algorithms such as BFS, IDS, BiBFS, and IDBiHS. The results demonstrate that our BF-enhanced bidirectional search offers superior memory usage performance with comparative performance in the number of node expansions and runtime, particularly in scenarios where traditional methods struggle with scalability. This paper provides a detailed analysis of our method, highlighting its advantages and potential applications in solving complex search problems.

2 Background and Related Work

Search algorithms can be classified into three main categories based on their memory usage:

- **Linear Memory (LM)**: These algorithms store only a single branch of the search tree (a path), resulting in memory consumption of $O(d)$, where d is the depth of the search.
- **Fixed Memory (FM)**: These algorithms are allocated a fixed amount of memory M (in addition to the memory needed for storing a single path) and must not exceed this limit. Categories 1 and 2 are collectively referred to as Restricted Memory (RM).

- **Unrestricted Memory (UM)**: These algorithms have no memory restrictions and typically use memory proportional to the size of the explored search tree.

2.1 Unidirectional Search Algorithms

Breadth-First Search (BFS), Depth-First Search (DFS), and Iterative Deepening Search (IDS) are foundational search algorithms, each with distinct characteristics suited to different types of problems. BFS is an uninformed UM search algorithm that systematically explores a graph in a level-order manner. By expanding all nodes at the current depth level before moving on to nodes at the next depth level, BFS ensures that the shortest path, in terms of edge count, is identified in an unweighted graph. The algorithm utilizes a queue-based data structure to manage the exploration process, where each node's neighbors are enqueued for subsequent examination. This structured approach guarantees that the first time a node is visited, it is reached through the shortest possible path, making BFS particularly well-suited for shortest-path discovery in unweighted graphs. Moreover, BFS is complete, meaning that if a solution exists, BFS is guaranteed to find it, since it exhaustively explores all possible paths in increasing order of length. This property and its ability to find the shortest path make BFS a robust choice for many search problems in unweighted graphs. In contrast, DFS is a LM search algorithm that explores a graph by diving deep along a single path until it reaches a dead end or the target node, at which point it backtracks to explore alternative paths. This approach can be implemented using either a stack-based data structure or recursion. DFS is generally more memory-efficient than BFS, as it does not need to store all nodes at the current depth level and instead stores only the current expanded branch. However, it has the drawback of potentially getting trapped in deep or infinite paths, making it less reliable for finding the shortest path in large or unbounded graphs. IDS combines the memory efficiency of DFS with the completeness of BFS. IDS achieves this by repeatedly applying DFS with an increasing depth limit, starting from a shallow depth and gradually extending the limit. This strategy allows the algorithm to prioritize and explore shallower solutions first, ensuring that it can find the optimal solution even when the depth of the solution is unknown. IDS is particularly advantageous in scenarios with a vast search space, as it balances the need for thorough exploration with memory constraints, making it space-efficient and complete.

2.2 Bidirectional Search

Unlike unidirectional search algorithms that focus on finding a path from the start node to the goal, bidirectional search operates by simultaneously initiating two searches: one forward from the start node and another backward from the goal node, with the intention of meeting in the middle. This approach significantly reduces the search space, particularly in large graphs, as each search only needs to traverse roughly half the distance, leading to a potentially exponential reduction in the number of nodes explored. The efficiency gains from this strategy are especially pronounced in problems with vast solution spaces, where the convergence of forward and backward searches can occur much faster than in traditional unidirectional searches. Bidirectional search can be implemented using various search algorithms, including BFS, DFS, and IDS, and more advanced algorithms developed to optimize the search process further. The flexibility to combine different search techniques in each direction makes bidirectional search a powerful tool for solving complex graph-based problems. General Breadth-First Heuristic Search (GBFHS) [1] is a UM bidirectional search algorithm that incrementally increases the search depth, utilizing a split func-

tion to determine the depth for each side. GBFHS expands all expandable nodes until a node is found in both open lists, with combined costs less than or equal to the current depth limit. This algorithm iteratively adjusts the depths until a solution is found, balancing exploration between the forward and backward searches. Iterative Deepening Bidirectional Heuristic Search (IDBiHS) [2] is a search algorithm that combines the features of iterative deepening search and bidirectional search. IDBiHS conducts depth-first search (DFS) iterations with thresholds similar to those used in GBFHS but operates within a LM constraint. IDBiHS finds a solution by iteratively increasing a threshold and employing split functions to determine the meeting points between forward and backward searches, thus ensuring efficient search space exploration.

2.3 Bloom Filter

Bloom filter is a memory-efficient data structure commonly used for membership queries in a given set [3]. The Bloom filter employs k hash functions, denoted as $h_1, h_2, ..., h_k$, to hash elements into an array of size m. To set up the Bloom filter, the array bits at positions $h_1(s), ..., h_k(s)$ are set to 1 for each element s in the set. Then, when checking for the presence of item q in the set, we examine the bits at positions $h_1(q), h_2(q), ..., h_k(q)$, if all the bits are set to 1, item q is considered to be present in the set. This approach comes with a small probability of producing a false positive error, which means that it may incorrectly indicate that an item is in the set when it is not. The probability of a false positive error depends on the selection of the parameters m (bloom filter size) and k (number of hash functions). After the insertion of n elements at random to the array of size m, the probability that a particular bit is 0 is precisely $(1 - \frac{1}{m})^{kn}$. Hence, the probability of a false positive error in this situation is $E_{FP} = ((1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k$.
On the other hand, the Bloom filter does not produce false negative errors, meaning that if the filter indicates that an element is not present, it is guaranteed to be absent from the set. This property makes Bloom filters particularly useful for applications where the guarantee of non-existence is critical, even at the expense of allowing a controlled probability of false positives. In addition, Bloom filters offer significant space savings compared to alternative data structures that provide exact membership testing at the cost of extensive memory usage.

3 Methodology

In this project, we propose BiBFIDS, a combination of Bloom filter's memory efficiency with the IDS algorithm's LM approach, along with the bidirectional search's fast convergence. Algorithm 1 presents a simple overview of the BiBFIDS algorithm. The algorithm defines two depth limits and repeats two symmetric phases: the forward search (lines 8-15) and the backward search (lines 17-24). Each search starts by increasing the other side's search depth limit (d2 is the depth limit of the backward search, and d1 is the depth limit of the forward search). After performing each search, if BiBFIDS obtains a candidate list, it attempts to find a solution by applying a depth-limited DFS from each candidate to the start/ goal node (depending on the search's side) using the depth limit of the other side (which guarantees that the solution length is d1+d2). Otherwise, if BiBFIDS obtains a bloom filter, it starts by finding candidate nodes that pass the filter and then continues with the depth-limited DFS as previously described. Finally, BiBFIDS constructs and returns its solution path if a candidate is found under both depth limits.

Algorithm 1 Bidirectional Bloom Filter based Iterative Deepening Search (BiBFIDS)

```
Require: S, G
if S = G then
    return [S]
end if

d1 ← 0
d2 ← 0
while true do
    d2 ← d2 + 1
    res ← search(S, G, d1, d2, 'F')
    if res is not None then
        solution_path ← get_solution_path(res, S, G, d1, d2, 'F')
        if solution_path is not None then
            return solution_path
        end if
    end if

    d1 ← d1 + 1
    res ← search(S, G, d1, d2, 'B')
    if res is not None then
        solution_path ← get_solution_path(res, S, G, d1, d2, 'B')
        if solution_path is not None then
            return solution_path
        end if
    end if
end while
```

3.1 Forward/Backward Search

Algorithm 2 presents the steps of each search (forward and backward). For each side (2-3 for forward and 5-6 for backward), the search begins by setting a bloom filter based on the frontier nodes (nodes at the depth limit) from the other side. For example, when applying the forward search, the bloom filter is defined by nodes at depth d2 from the goal node (the frontier). Then, BiBFIDS attempts to find candidate nodes from the current side (i.e., the side of the search) at the corresponding depth limit. If the get_candidates algorithm (Algorithm 3) returns a list of node candidates, it is returned to the main algorithm (Algorithm 1). Otherwise, if too many candidates are found (more than 100), BiBFIDS obtains a new bloom filter, which is set by all the found candidates, and we proceed with minimizing the candidate list.

Algorithm 2 Search

```
Require: S, G, d1, d2, side
 1: if side == 'F' then
 2:     bf ← set_bloom_filter_by_depth(G, d2, 'B')
 3:     res ← get_candidates(bf, S, d1, 'F')
 4: else
 5:     bf ← set_bloom_filter_by_depth(S, d1, 'F')
 6:     res ← get_candidates(bf, G, d2, 'B')
 7: end if
 8:
 9: if res is a BloomFilter then
10:     other_bf ← res
11: else
12:     return res
13: end if
14:
15: return get_optimized_candidates(S, G, d1, d2, other_bf, side)
```

3.2 Getting the Candidate List

Algorithm 3 presents BiBFIDS's process of finding possible candidate nodes. We define a candidate node as a node that appears in the bloom filter set by the frontier nodes from the other side). To identify all the candidate nodes, BiBFIDS utilizes a limited DFS to arrive at the current side's frontier nodes. Then, for each frontier node, BiBFIDS checks if it passes the bloom filter defined by the other side (line 9). If the node does not pass the filter, we obtain a guarantee that this node can not connect both sides of the search. This can be guaranteed due to Bloom filter's property of zero false positive error, i.e., if a node was used to set the filter, it is guaranteed to pass the filter. Otherwise, if a node passes the filter, it is added to the candidate list. Note that to utilize the memory efficiently, we limit the length of the candidate list to 100. In cases where BiBFIDS identifies more than 100 candidates, we set a new bloom filter based on the candidates (unlike the previous filter, which was set by the frontier nodes of the other side). Finally, we return the candidate list (if less than 100 candidates) or the bloom filter (if more than 99 candidates).

Algorithm 3 Get Candidates

Require: bf, node, max_depth, side

```
stack ← [node]
candidates ← []
other_bf ← None
max_length ← 100

while stack is not empty do
    state ← stack.pop()
    if (side = 'F' and state.gF = max_depth) or (side = 'B' and state.gB = max_depth) then
        if bf.check(state) then
            if len(candidates) < max_length then
                candidates.append(state)
            else
                if other_bf is None then
                    other_bf ← BloomFilter()
                    for candidate in candidates do
                        other_bf.add(candidate)
                    end for
                end if
                other_bf.add(state)
            end if
        else
            for neighbor in state.get_neighbors() do
                if (side == 'F' and neighbor.gF ≤ max_depth) or (side == 'B' and neighbor.gB ≤
                max_depth) then
                    stack.append(neighbor)
                end if
            end for
        end if
    end if
end while
if other_bf is None then
    return candidates
else
    return other_bf
end if
```

3.3 Optimizing the Candidate List (Ping-Pong)

In case the get_candidates algorithm (Algorithm 3) returns a bloom filter, it means too many candidates were found (more than 99). Algorithm 4 presents BiBFIDS's candidate list optimization process. This optimization process is done by attempting to find candidates from the other side that also pass the bloom filter set by the candidate list from the current side. If less than 100 candidates were found, it means that BiBFIDS successfully optimized the candidate list, and it is returned to the main algorithm (Algorithm 1). Otherwise, if another bloom filter is returned, BiBFIDS compares the number of inserted nodes to each filter. If the new filter has more insertions than the previous one, it means we already have an optimized filter, and we return it to the main algorithm. However, if the new filter has fewer insertions than the previous one, it means that we can minimize the filter further, which is done by repeating the process also for the second side (optimizing the optimized filter) to optimize the outcome further.

Algorithm 4 Get Optimized Candidates

Require: S, G, d1, d2, bf, side

```
1:  if side == 'F' then
2:      while true do
3:          res ← get_candidates(bf, G, d2, 'B')
4:          if res is a BloomFilter then
5:              other_bf ← res
6:          else
7:              return res
8:          end if
9:
10:         if bf.number_insertions ≤ other_bf.number_insertions then
11:             return bf
12:         end if
13:
14:         res ← get_candidates(other_bf, S, d1, 'F')
15:         if res is a BloomFilter then
16:             bf ← res
17:         else
18:             return res
19:         end if
20:
21:         if other_bf.number_insertions ≤ bf.number_insertions then
22:             return other_bf
23:         end if
24:     end while
25: else
26:     Symmetrical for backward search.
27: end if
```

4 Evaluation

4.1 Evaluation Settings

We evaluated BiBFIDS on three well-known combinatorial games: the Pancake problem, the Sliding-Tile puzzle, and the Top-Spin puzzle, each presenting unique challenges and characteristics that test the efficacy of various search algorithms. These games are classic combinatorial challenges used to evaluate search algorithms. The Pancake problem involves sorting a stack of pancakes by size using a series of flips, with rising complexity based on the problem's size. The Sliding-Tile puzzle requires arranging tiles on a grid in numerical order by sliding them into an empty space, posing a challenge due to its ample search space. The Top-Spin puzzle, a circular variation of the Pancake problem, involves flipping a subset of disks to achieve a numerical sequence, adding complexity through its cyclic structure. We examined three difficulty levels for each game: small, medium, and large. In the Pancake problem, we used 5, 7, and 9 pancakes, respectively. In the Sliding-Tile puzzle, we used 3x3, 5x5, and 7x7 grids, respectively. In the Top-Spin puzzle, we used 10, 15, and 20-length arrays, respectively. We compared our method to several established search algorithms, including BFS, DFS, IDS, bidirectional BFS (BiBFS), and IDBiHS (with a constant 0 heuristic function for fair comparison). The evaluation was performed over 100 random initial states for each problem and size and focused on three key metrics: the mean number of node expansions (and their percentage of all encountered nodes), the mean runtime, and the mean memory usage required from each algorithm to find a solution path. The number of node expansions measures an algorithm's efficiency in search space exploration, with fewer expansions indicating a more effective process. Runtime measures an algorithm's required time to return a solution path, and memory usage measures how much memory the algorithm requires to run. These measurements are crucial when only limited resources can be utilized for the search algorithm. By analyzing these metrics across different algorithms and problem domains, we provide a comprehensive comparison that highlights the strengths and weaknesses of each approach and offers insights into their suitability for various types of search problems.

4.2 Evaluation Results

Table 1 presents the results of our evaluation. The domain column describes which problem domain is used for evaluation (sliding tile, topspin, and pancakes). The algorithm column describes which algorithm is being evaluated (BFS, IDS, BiBFS, IDBiHS, and the BiBFIDS variants). Note that we evaluate three variants of the BiBFIDS algorithm (10K, 100K, and 1M), with each variant having a different size for its bloom filters according to the variant name (i.e., BiBFIDS 10K utilizes bloom filters with 10K bits). We use those three variants to evaluate how the bloom filter size affects BiBFIDS's memory usage and runtime. Each algorithm was evaluated on three sizes of each domain (small, medium, and large) to evaluate how its performance is affected by smaller problems (i.e., smaller search space) and larger problems (i.e., larger search space).

As can be seen in Table 1, when comparing the algorithms on small problems, the BiBFIDS algorithm (including all its variants) constantly achieves the second-best performance regarding the number of expansions independently of the bloom filter's size. This observation can be explained by the fact that in smaller problems, the solution paths are usually shorter. Therefore, fewer nodes need to be expanded to find the solution, which might not require a large filter, such as in the 100K and 1M variants, and a 10K filter (or maybe smaller) is sufficient to find the solution quickly. The

Domain	Algorithm	Small			Medium			Large		
		Expansions	Runtime	Memory Usage	Expansions	Runtime	Memory Usage	Expansions	Runtime	Memory Usage
Sliding Tile	BFS	164.23 (56%)	0.018	0.115	239.53 (56%)	0.067	0.239	108.8 (55%)	0.063	0.161
	IDS	11402.2 (94%)	0.426	0.005	39805.12 (95%)	3.561	0.008	5001.72 (95%)	0.947	0.011
	BiBFS	**25 (46%)**	**0.001**	0.015	**29.36 (47%)**	**0.001**	0.024	**19.48 (46%)**	**0.001**	0.024
	IDBiHS	3421.54 (37%)	0.089	**0.003**	10683.63 (38%)	0.407	**0.006**	1406.9 (38%)	0.063	**0.008**
	BiBFIDS (10K)	174.8 (45%)	0.035	0.004	275.77 (45%)	0.085	**0.006**	118.39 (44%)	0.057	**0.008**
	BiBFIDS (100K)	174.8 (45%)	0.038	0.015	275.77 (45%)	0.078	0.016	118.39 (44%)	0.054	0.018
	BiBFIDS (1M)	174.8 (45%)	0.039	0.128	275.77 (45%)	0.077	0.129	118.39 (44%)	0.061	0.13
TopSpin	BFS	1644.59 (48%)	0.188	1.481	1647.19 (48%)	0.277	1.741	3493.26 (49%)	0.723	4.093
	IDS	3069.11 (98%)	0.165	0.006	4023.24 (98%)	0.276	0.007	7283.51 (99%)	0.588	0.009
	BiBFS	**52.7 (43%)**	**0.002**	0.042	**51.5 (43%)**	**0.002**	0.046	**75.73 (45%)**	**0.003**	0.074
	IDBiHS	2973.26 (43%)	0.092	0.006	2513.03 (44%)	0.086	0.007	5361.49 (45%)	0.182	0.009
	BiBFIDS (10K)	152.1 (47%)	0.027	**0.005**	210.44 (48%)	0.039	**0.005**	384.62 (49%)	0.065	**0.007**
	BiBFIDS (100K)	152.1 (47%)	0.03	0.016	149.12 (47%)	0.033	0.016	226.21 (49%)	0.051	0.017
	BiBFIDS (1M)	152.1 (47%)	0.027	0.128	149.12 (47%)	0.034	0.129	226.21 (49%)	0.053	0.13
Pancakes	BFS	60.62 (59%)	0.005	0.033	1976.65 (52%)	0.271	1.307	98967.06 (44%)	19.019	76.592
	IDS	175.73 (95%)	0.007	0.004	9696.47 (99%)	0.656	0.01	2687063.18 (100%)	251.365	0.027
	BiBFS	**12.81 (28%)**	**0.001**	0.013	**93.36 (20%)**	**0.009**	0.16	**897.9 (15%)**	**0.716**	2.178
	IDBiHS	60.74 (29%)	0.003	**0.003**	3168.8 (20%)	0.231	**0.007**	383987.54 (14%)	49.806	**0.017**
	BiBFIDS (10K)	27.63 (35%)	0.008	0.004	538.05 (32%)	0.128	0.011	2135836.37 (84%)	196.175	0.072
	BiBFIDS (100K)	27.63 (35%)	0.009	0.015	215.53 (30%)	0.089	0.018	28495.64 (31%)	3.893	0.025
	BiBFIDS (1M)	27.63 (35%)	0.009	0.127	215.53 (30%)	0.089	0.13	2586.13 (28%)	1.189	0.135

Table 1: Number of expansion, runtime, and memory usage of each algorithm when applied to various domains and sizes. The best performance appears in **Bold** and the second-best in underlined.

explanation can be validated by observing the memory usage results in small problems. In these cases, the 10K variant of BiBFIDS consistently achieves the best or second-best performance, while the other variants require more memory even though they expand the same amount of nodes.

When observing the results on the medium problems, we can see that the 10K variant of BiBFIDS starts to fall behind the other variants, which as previously explained, might be due to the problem sizes and longer solution paths that require more node expansions compared to the small problems. As can be seen, BiBFIDS keeps achieving the second-best performance in the number of expansions in both the topspin and pancakes problems, achieving relatively close results to the second-best in the sliding tile problem. Additionally, we can observe that by increasing the problem size, the other algorithms also increased their memory usage while the BiBFIDS variants maintained their low memory usage, achieving second-best performance in both the sliding time and topspin problems and a close result to the second-best in the pancakes problem.

After evaluating both small and medium problems, we continue to evaluate larger problems. In these problems, each algorithm is required to expand more nodes, which increases the number of expansions, which is significantly visible for the pancake problem. This behavior can be explained by the operator definitions in the pancake problem, which allow flipping any number of pancakes, from two to the entire stack. The larger the pancake problem, the more operators there are as there are more options to flip different amounts of pancakes (in fact, for problem size K, there are always K-2 operators as we do not allow to flip a single pancake and we do not allow for a node to travel back to its parent). As can be seen, by the results, BiBFIDS maintains its optimal memory usage in both the sliding tile and topspin puzzles and, in general, maintains the same amount of memory usage, unlike other algorithms such as BFS and BiBFS, which require more memory than the larger the problem becomes.

5 Discussion

5.1 BiBFIDS vs. IDBiHS

Figure 1 shows the mean memory usage of the evaluated algorithms on Pancake problems with varying sizes. As can be seen, for small problems (size of 5), the value of using BiBFIDS is insignificant due to the BF's size, which requires a significant portion of the memory. However, the memory usage remained relatively constant for larger problems, which makes BiBFIDS highly scalable for big problems. While IDBiHS achieves similar results for problems smaller or equal to 8, BiBFIDS does come out on top for larger problems (as evident for size 9). In addition, as can be seen in Table 1, BiBFIDS outperforms IDBiHS's runtime in most cases, which allows for better trade-off between memory usage and runtime as BiBFIDS requires comparable memory usage to IDBiHS.

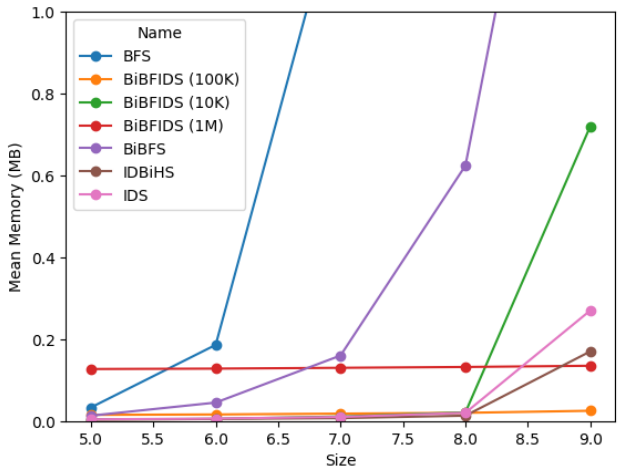

Figure 1: Mean memory usage of the evaluated algorithms on Pancake problems with varying sizes.

5.2 Heuristic Search

In this project, we did not explore incorporating heuristics into BiBFIDS. However, this future work direction can be easily incorporated into the algorithm by utilizing heuristic-based value limits (as used in the GBFHS and IDBiHS algorithms) instead of the current depth limits. Then, instead of expanding nodes based on their depth, BiBFIDS would expand nodes based on their h/f values, which could further optimize all three measures (number of node expansions, runtime, and memory usage) of the BiBFIDS algorithm.

6 Conclusions

In this project, we presented BiBFIDS, a combination of BF's memory efficiency with the IDS algorithm's LM approach, along with the bidirectional search's fast convergence. As evidenced by the evaluation results, although BiBFIDS's algorithm is simple, it efficiently utilizes memory to find solutions for various problems, even when the problem size is large. Compared to its main rival algorithm, the IDBiHS algorithm, BiBFIDS utilizes a similar amount of memory (sometimes even less) and converges faster to find a solution, as evidenced by the runtime result comparison. Future work could include further optimizing the BiBFIDS by dynamically changing the bloom filter size to be independent of the problem's size. An additional direction could be incorporating a heuristic search into BiBFIDS to minimize further its number of expansions, runtime, and memory usage.

References

[1] M. Barley, P. Riddle, C. L. López, S. Dobson, and I. Pohl, "Gbfhs: A generalized breadth-first heuristic search algorithm," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 9, no. 1, 2018, pp. 28–36.

[2] S. S. Shperberg, S. Danishevski, A. Felner, and N. R. Sturtevant, "Iterative-deepening bidirectional heuristic search with restricted memory," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, 2021, pp. 331–339.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.