b>

Yes, my code generates the same output for each public test case, but this may not be the case in general because, IDS and Bidirectional Breadth-First Search differ in exploration strategies, and hence have the potential to result in different outputs. IDS works by making a number of depth-limited searches that incrementally increase the limit on depth. This is complete and optimal but potentially inefficient because explorations of nodes at shallower depths can result in repeated locations. However, Bidirectional BFS explores both from the start and the goal simultaneously and can find a solution much faster than the unidirectional version if they happen to meet in the middle. It can be memory-intensive since it has nodes from both directions in memory. The reason is that IDS may fail to find an optimal solution when the depth limit is small, while Bidirectional BFS might be inefficient if the meeting point is deep or it suffers from heavy memory usage. Here, which one will be used in practice better depends on what the structure of the search space is and on the cost or depth of the solution.

c>

The IDS algorithm is unfortunately very slow, taking **20 minutes and 48.3 seconds** to complete, but it is highly memory-efficient at **0.37 MB**. This aligns with the nature of the IDS strategy, which uses depth-limited searches that conserve memory but are computationally intensive due to redundant node exploration.

In contrast, the BBDS algorithm performs significantly faster, completing the search in just 19.2 seconds while consuming minimal memory (0.0001 MB). The bidirectional approach of BBDS enhances performance by reducing the number of nodes explored, although it may not be applicable in all problem scenarios due to specific constraints.

e>
In A*, it used **1 minute and 35.7 seconds**, consuming **0.625 MB** in memory usage. Even though this algorithm is slower than Bidirectional Heuristic Search, this is faster than Iterative Deepening Search. Additionally, it takes longer time to compute since it utilises the heuristic approach, which needs many paths to be tested before proceeding towards the goal.

On the other side, the Bidirectional A* that used to generate the best performance. The task was completed in **1 minute and 2.4 seconds**. It consumed less memory at only **0.11 MB**. This efficiency is realised through a combination of heuristic guidance and the fact that it is possible to begin the search simultaneously from the start and the goal states. The above approach will most optimally balance execution speed and memory usage.

No, my code gives different output for different cases and let see some example case

Enter the start node: 4
Enter the end node: 12
A* Path: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12]
Bidirectional Heuristic Search Path: [4, 34, 33, 11, 32, 31, 3, 5, 97, 28, 10, 12]

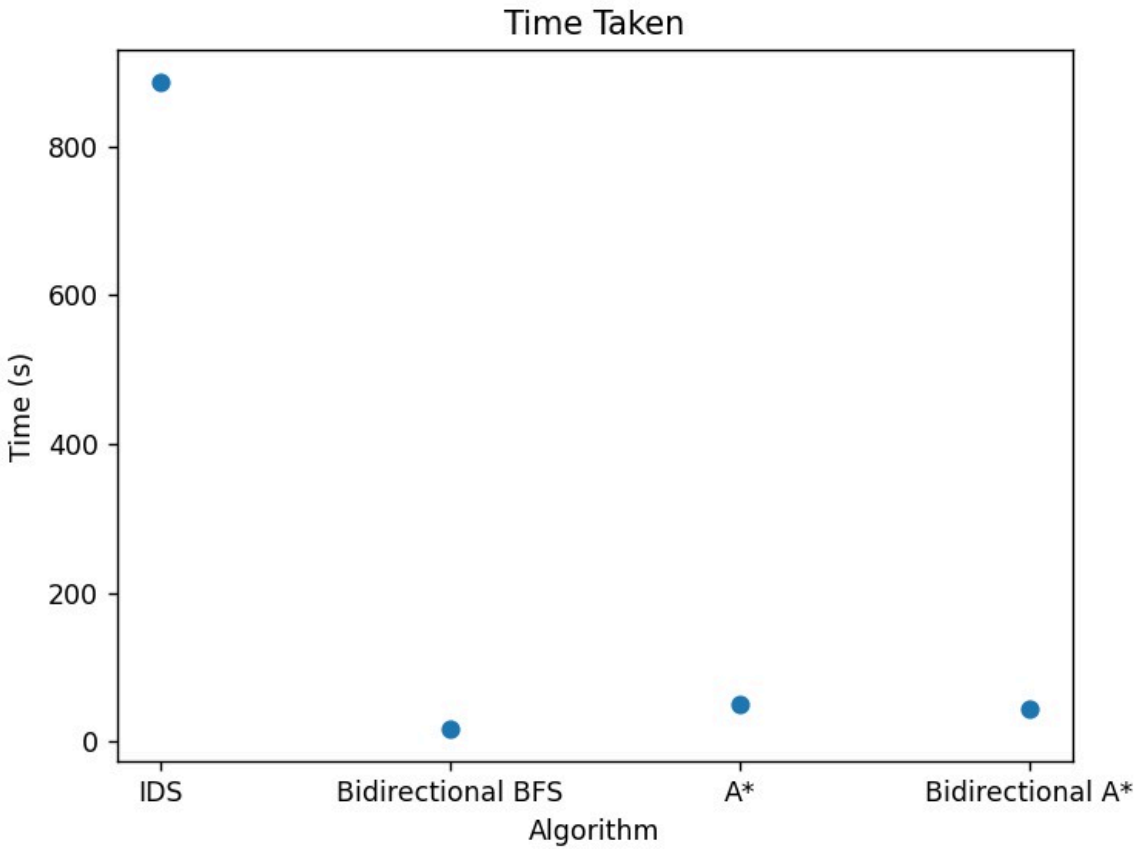
This is because A* and Bidirectional A* differ in their performance owing to the exploration strategies and memory requirements. A* Search uses a single front for expanding nodes; this is based on the combined cost function f(n) = g(n) + h(n), where g(n) stands for the cost from the start node and h(n) is the heuristic estimate to the goal. The approach ensures it finds a shortest path efficiently but relies on an admissible heuristic; however, this approach could be memory-intensive because all nodes are stored in the open and closed lists. Bidirectional A* Search improves this with simultaneous searches starting from the start and goal nodes, which reduces the numbers required to explore by perhaps halving them, depending on the points where the searches meet each other. This, however, makes it yet more memory-intensive because of the presence of two lists and requires heavy coordination for finding the intersections between the two searches. Hence, although Bidirectional A* might cut through the search space faster in some instances, its performance is critical to intersection management as well as has the drawback of increased memory usage, which A* handles in a much more straightforward manner, although possibly less efficient in terms of memory.

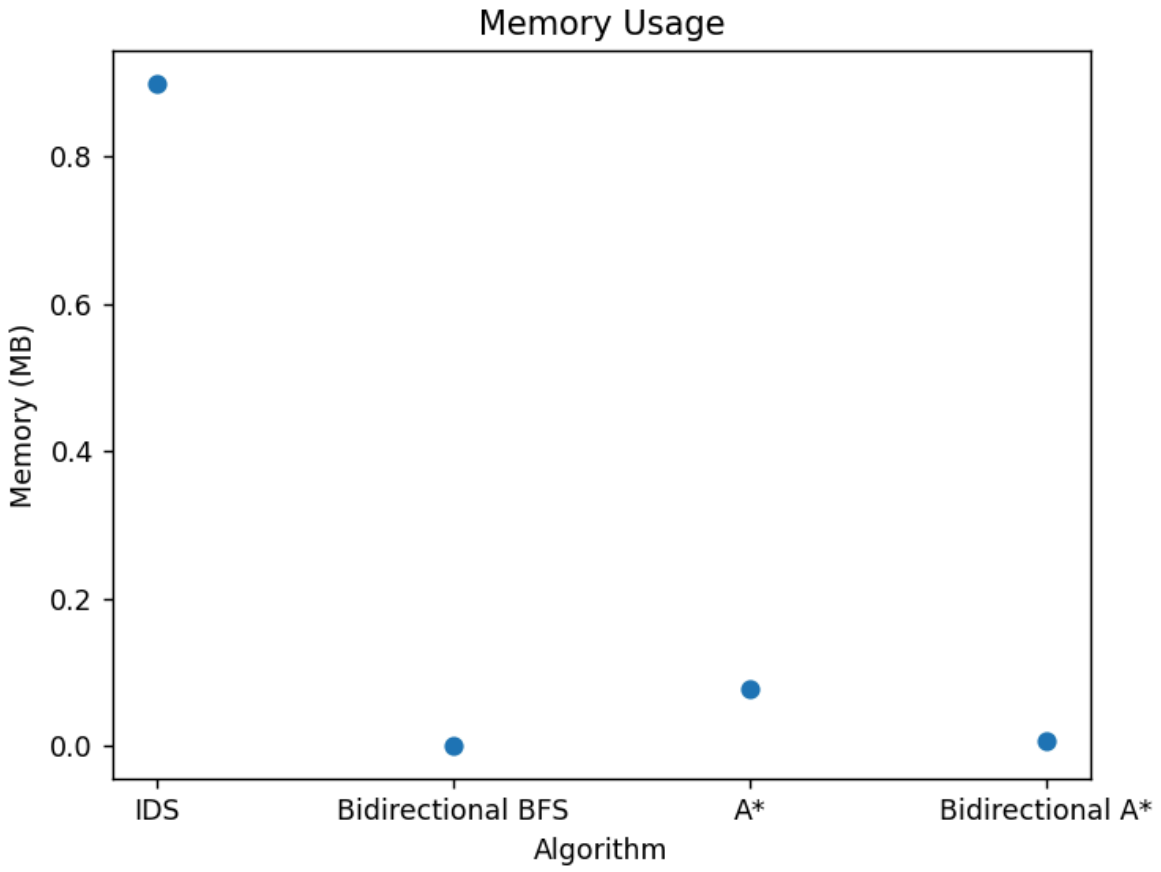
f>

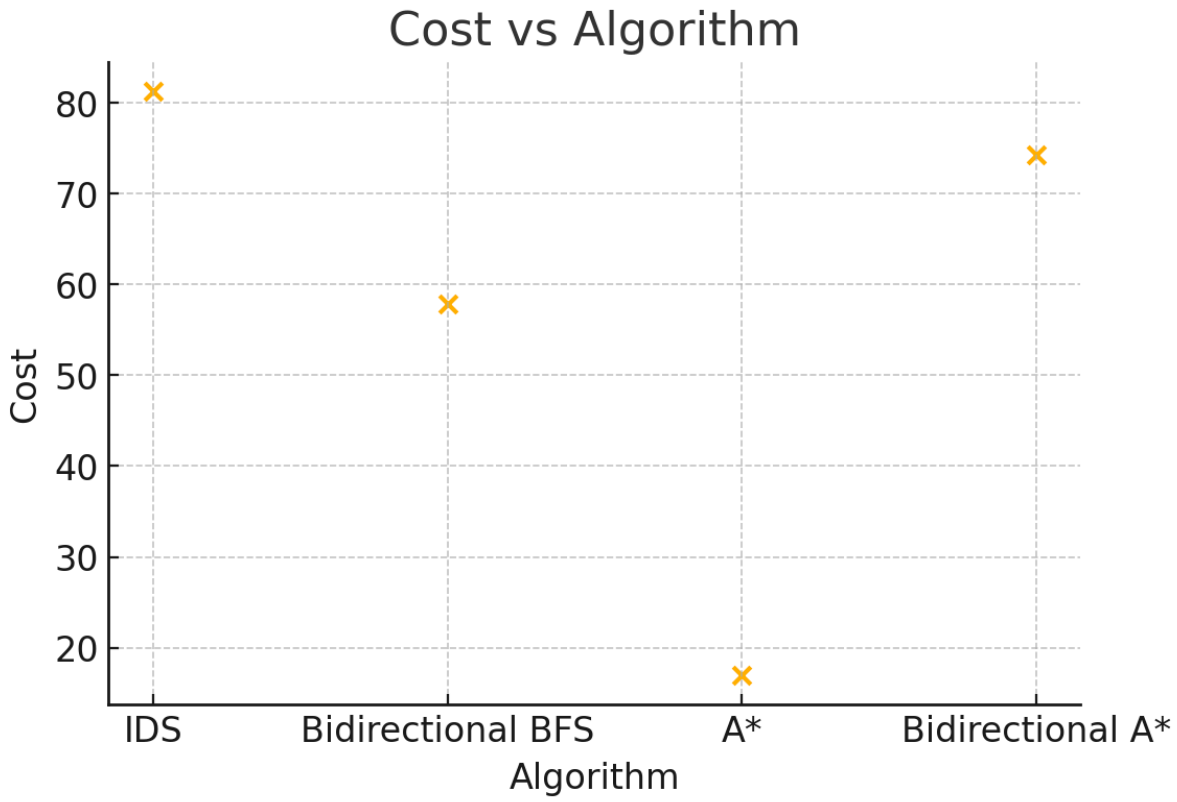
Time and Space Complexity

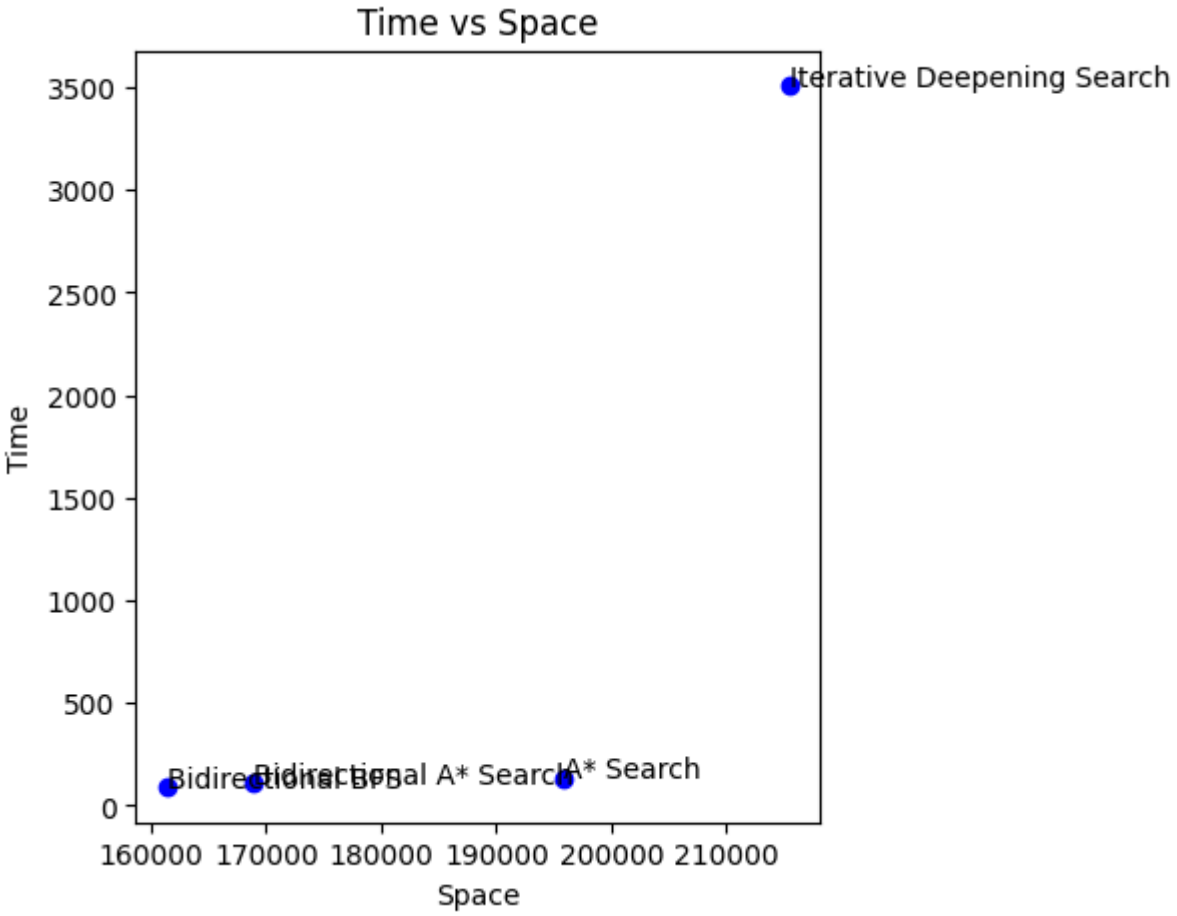
1. **Iterative Deepening Search (IDS)**
    - **Time Complexity**: O(b^d), where b is the branching factor and d is the depth of the solution. IDS performs a series of depth-limited searches, leading to a time complexity that is similar to BFS but with repeated node explorations.
    - **Space Complexity**: O(b*d). IDS only needs to store nodes on the current path and the nodes at the current depth, which is linear with respect to the depth.
2. **Bidirectional Breadth-First Search (BBDS)**
    - **Time Complexity**: O(b^(d/2)), where b is the branching factor and d is the depth of the solution. BBDS can potentially halve the effective depth of the search space due to searching from both ends.
    - **Space Complexity**: O(b^(d/2)). Memory usage is significant because it needs to store all nodes in the frontier for both the forward and backward searches.

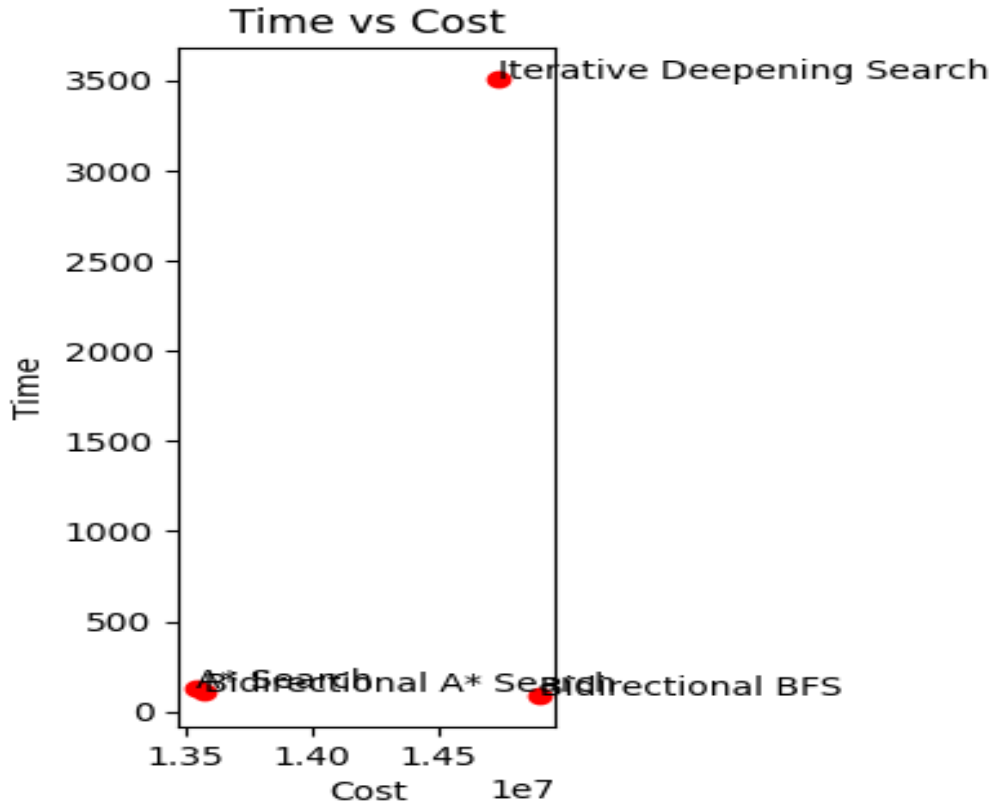
3. ***A Search****
    - **Time Complexity**: O(b^d)in the worst case, similar to IDS, but depends heavily on the heuristic function. With a good heuristic, the effective time complexity can be much lower.
    - **Space Complexity**: O(b^d). A* needs to store all nodes in the open and closed lists, which can be substantial depending on the depth and branching factor.
4. ***Bidirectional A* Search***
    - **Time Complexity**: O(b^(d/2)), similar to BBDS, but with the additional complexity of maintaining two open lists and performing heuristic evaluations from both directions.
    - **Space Complexity**: O(b^(d/2)), Requires storage for both forward and backward searches, similar to BBDS but often more efficient due to heuristics.

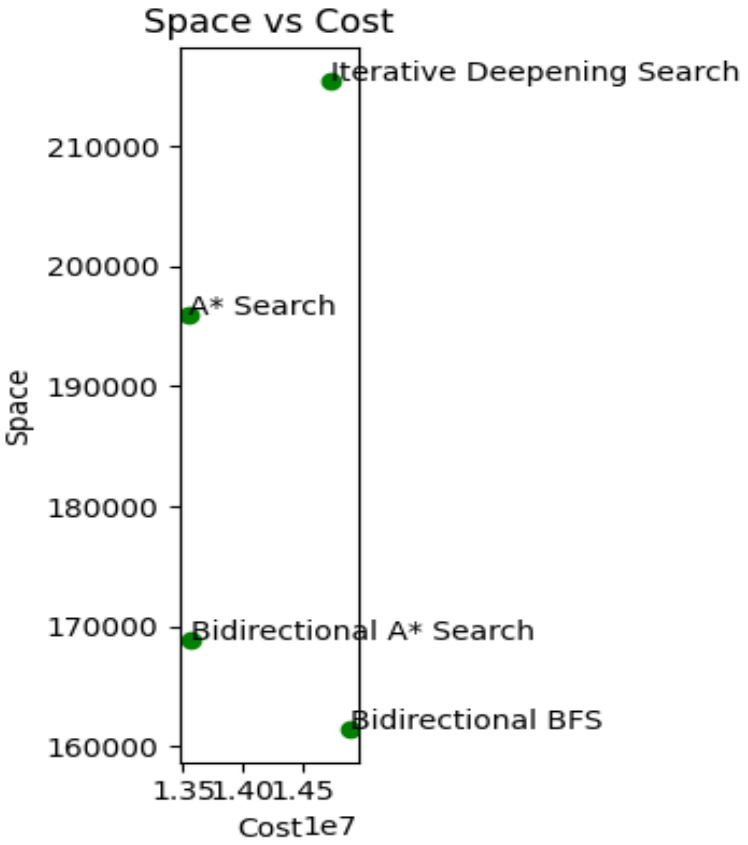


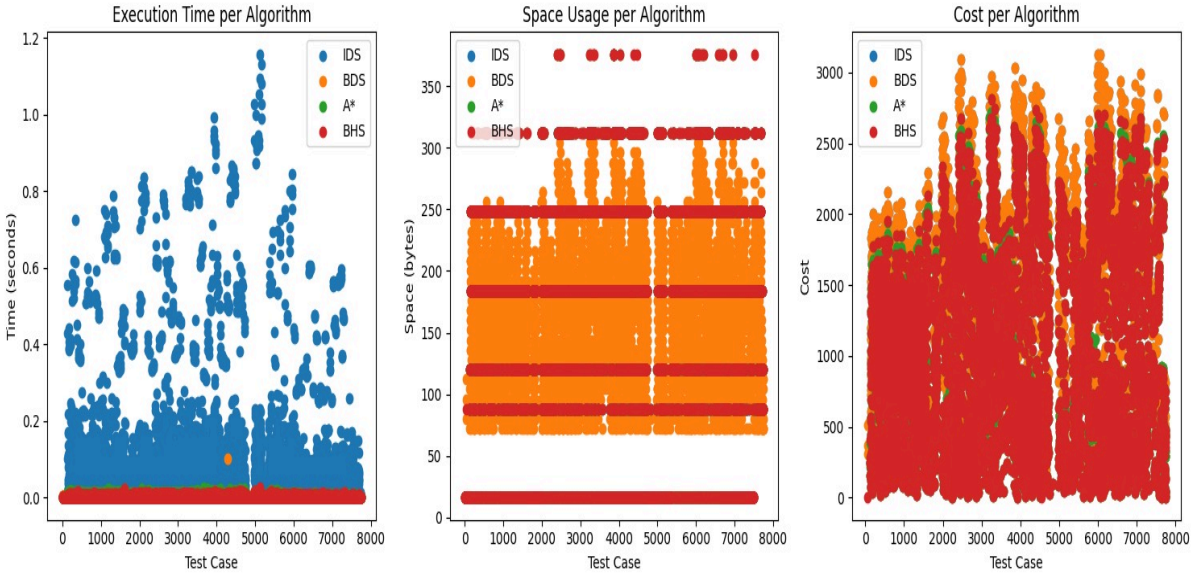

Memory Usage
Memory (MB)
0.8
0.6
0.4
0.2
0.0
IDS
Bidirectional BFS
A*
Bidirectional A*
Algorithm

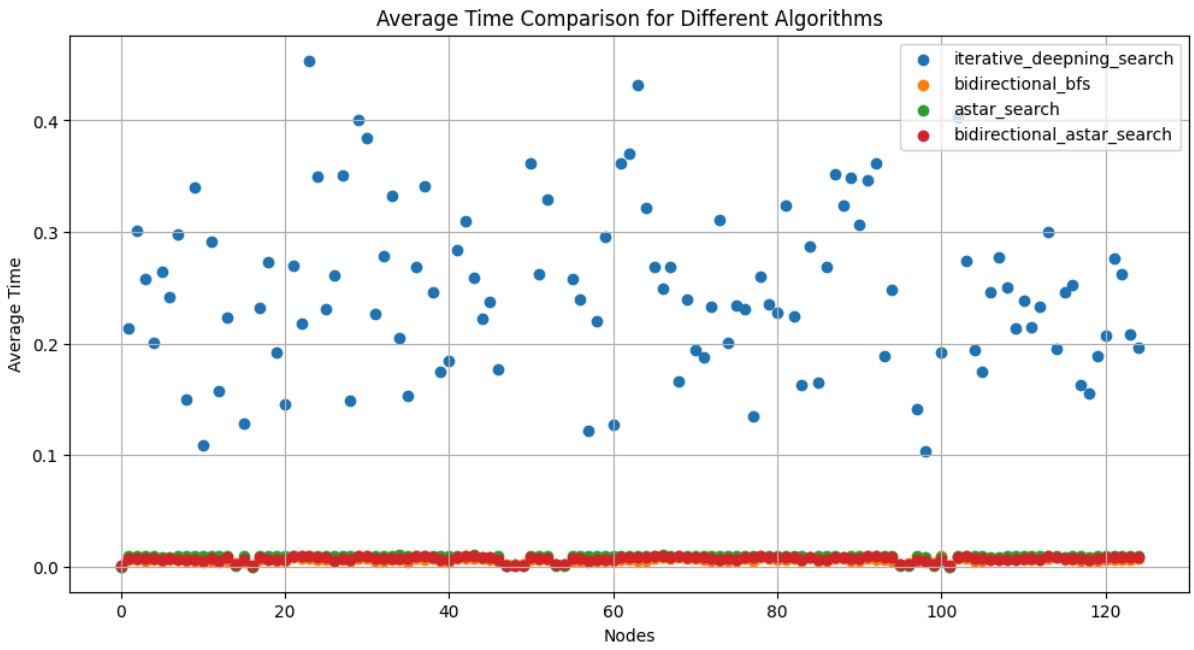

Cost vs Algorithm
Cost
80
70
60
50
40
30
20
IDS
Bidirectional BFS
A*
Bidirectional A*
Algorithm

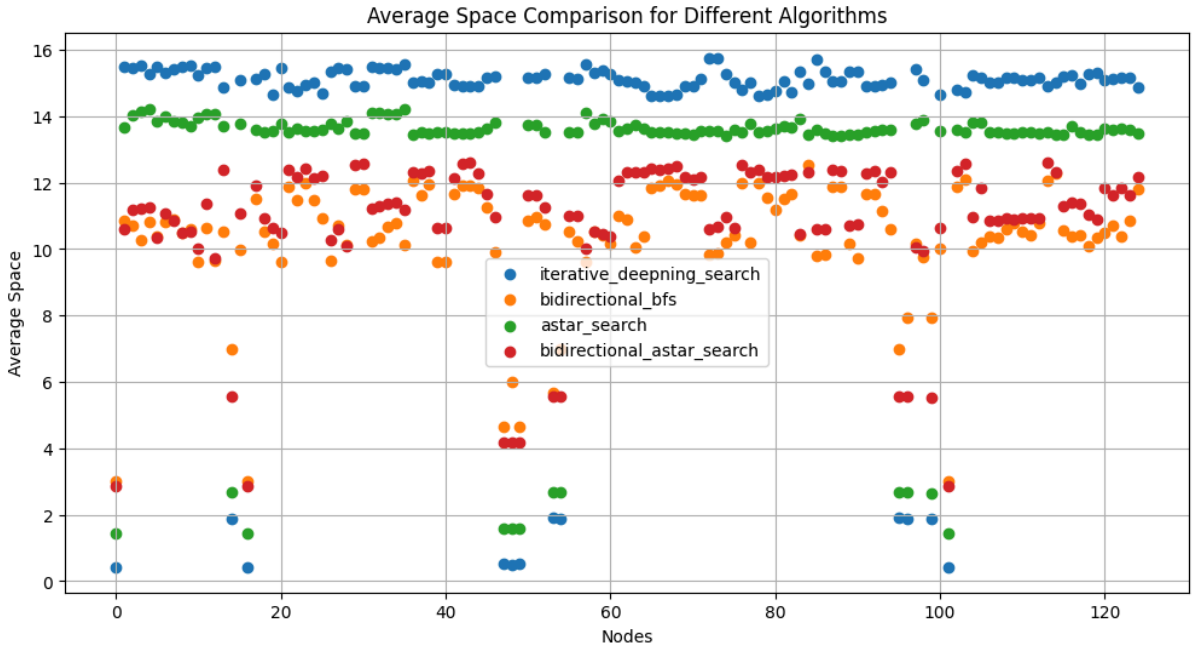

Time vs Space
Iterative Deepening Search
A* Search
Time
Space
3500
3000
2500
2000
1500
1000
500
0
160000
170000
180000
190000
200000
210000

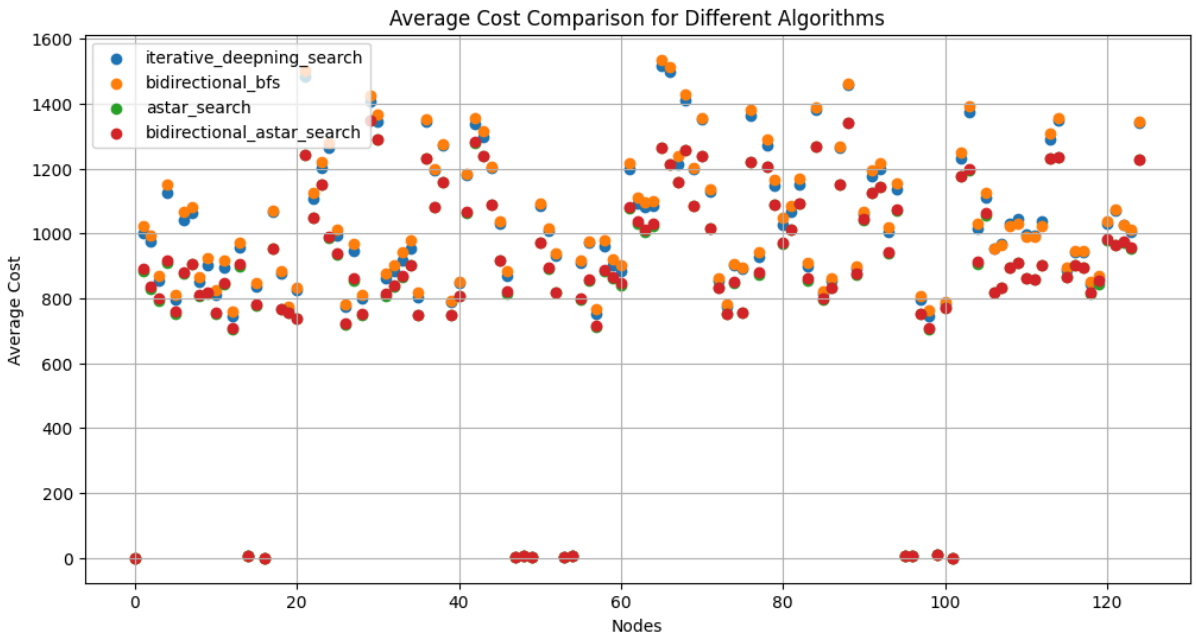

Time vs Cost
Iterative Deepening Search
Bidirectional BFS
Time
Cost
1e7
3500
3000
2500
2000
1500
1000
500
0
1.35
1.40
1.45


Space vs Cost
Iterative Deepening Search
A* Search
Bidirectional A* Search
Bidirectional BFS
Space
210000
200000
190000
180000
170000
160000
1.35
1.40
1.45
Cost
1e7


Execution Time per Algorithm
Time (seconds)
Test Case
Space Usage per Algorithm
Space (bytes)
Test Case
Cost per Algorithm
Cost
Test Case
IDS
BDS
A*
BHS


Average Time Comparison for Different Algorithms
iterative_deepning_search
bidirectional_bfs
astar_search
bidirectional_astar_search
Average Time
Nodes
0.0
0.1
0.2
0.3
0.4
0
20
40
60
80
100
120


Average Space Comparison for Different Algorithms
iterative_deepning_search
bidirectional_bfs
astar_search
bidirectional_astar_search
Average Space
Nodes
0
2
4
6
8
10
12
14
16
0
20
40
60
80
100
120



Brief Description of Algos

Merging the best of DFS with that of BFS is called Iterative Deepening Search (IDS). IDS uses both together to maintain an increasing depth limit instead of a queue as in BFS or a stack as in DFS. A search up to a depth limit of 0 would be a DFS. The goal might not have been found. Then, it was increased some amount, the search was repeated from scratch, and it proceeded in this way until the goal was found. Incremental depth improvement The sum of completion and optimization of IDS is obtained in increments when its depth is enhanced. Besides, in comparison with BFS, IDS requires less memory. However, in the worst-scenario scenario, it may become inefficient due to repeated explorations at shallower depths.

The Bidirectional Breadth-First Search (BBDS) increases the efficiency of searching. It does not go both ways from the source and destination nodes as in the case of BBDS but instead, it performs two breadth first searches that go away in opposite directions. Ideally, they should meet somewhere in between. This approach reduces the size of the search space and, in most cases, finds solutions faster than a traditional breadth-first search. However, it does require a lot of memory to accommodate nodes from both search fronts and is quite clumsy to implement when coordinating the search.

A* Here, we have an informed search algorithm in which the cost to get to the node (g(n)) is combined with a heuristic estimate of the cost to the goal (h(n)) in determining the nodes' priority. It expands nodes according to the function f(n) = g(n) + h(n), combining both these costs toward a more accurate move toward the goal. A* is complete and optimal if the heuristic is admissible, but very memory-intensive because it has to store all nodes both in open and closed lists.

Bidirectional A* extends A* Search to do two simultaneous searches, one from a start node and another from a goal node. Both the searches use the strategy of A* to find a meeting point; most of the times, this approach is expected to be much faster than single-direction A* while finding a solution. This method balances the efficiency gains of bidirectional search with heuristic guidance of A*, but it requires managing and coordinating two sets of nodes that could become complex and memory-intensive.