CSE 643: Artificial Intelligence

Assignment 1: Report

Aditya Aggarwal **2022028**

PART B

After running the Iterative Depth First Search algorithm and the Bidirectional Breadth-First Search algorithm for the given pairs of nodes present in the adjacency matrix, all the paths for public test cases are the same for both algorithms.

However, the output from these algorithms may be different when multiple paths exist with the shallowest depth from start_node to goal_node. In such a case, the IDS algorithm might choose the path that comes first in the adjacency matrix, whereas Bi-BFS may choose the path for which the intersection node is first met.

PART C

Time Complexity: The IDS algorithm takes exponentially larger time in comparison to the Bi-BFS algorithm. This is because the former searches all combinations of possible paths to goal_node for a specific depth before moving to the next depth. For example, for a solution with the shallowest depth of 30, the algorithm tries all path combinations for depth = 1, 2, 3, and so on till depth = 29. In contrast, the Bi-BFS algorithm optimizes the solution by reducing the time complexity to O(b^d/2).

Memory Usage: The IDS algorithm is a recursive algorithm that needs to store a path until it finds the solution for the given depth or discards it. Due to this nature, it consumes more memory than the Bi-BFS algorithm.

The above claims agree with the empirical results obtained by running both algorithms and computing their time and memory usage. These metrics are later represented in the scatter plot.

PART E

Repeating PART B for Heuristic algorithms:

The paths obtained from the A* search algorithm and Bidirectional Heuristic Search algorithm are the same, as they follow the same heuristic function:

$$h(w) = dist(u, w) + dist(w, v)$$

where *h(w)* represents the heuristic function, *w* represents the node, *u* represents the source node, and *v* represents the destination node.

Repeating PART C for Heuristic algorithms:

The time complexity and memory usage in these algorithms are roughly the same, with the Bidirectional Heuristic having slightly better memory usage and time consumption due to its bidirectional nature. However, that does not provide it a huge advantage over the A* algorithm because the intersection of nodes is not a valid termination condition in this case. This is because while the intersection provides a valid path from the start_node to the goal_node, it does not guarantee an optimal path.

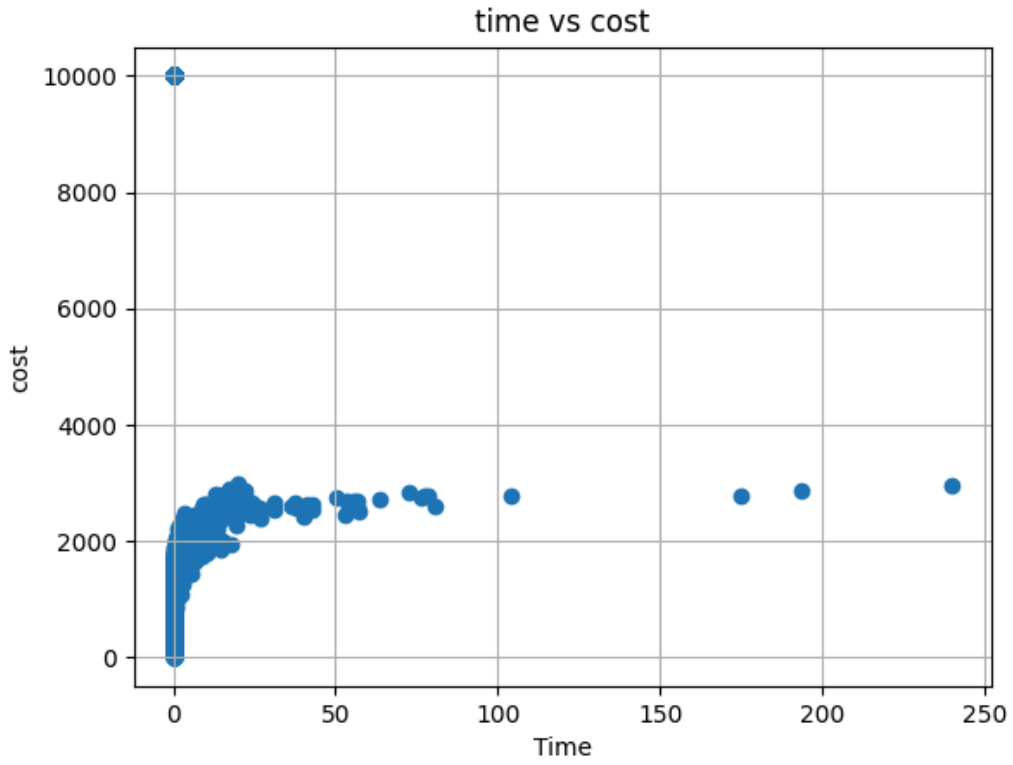
NOTE: The Bi-directional Heuristic Search algorithm implemented in the code follows the pseudo-code provided in the book. For reference, a screenshot of the same pseudo-code is provided below:

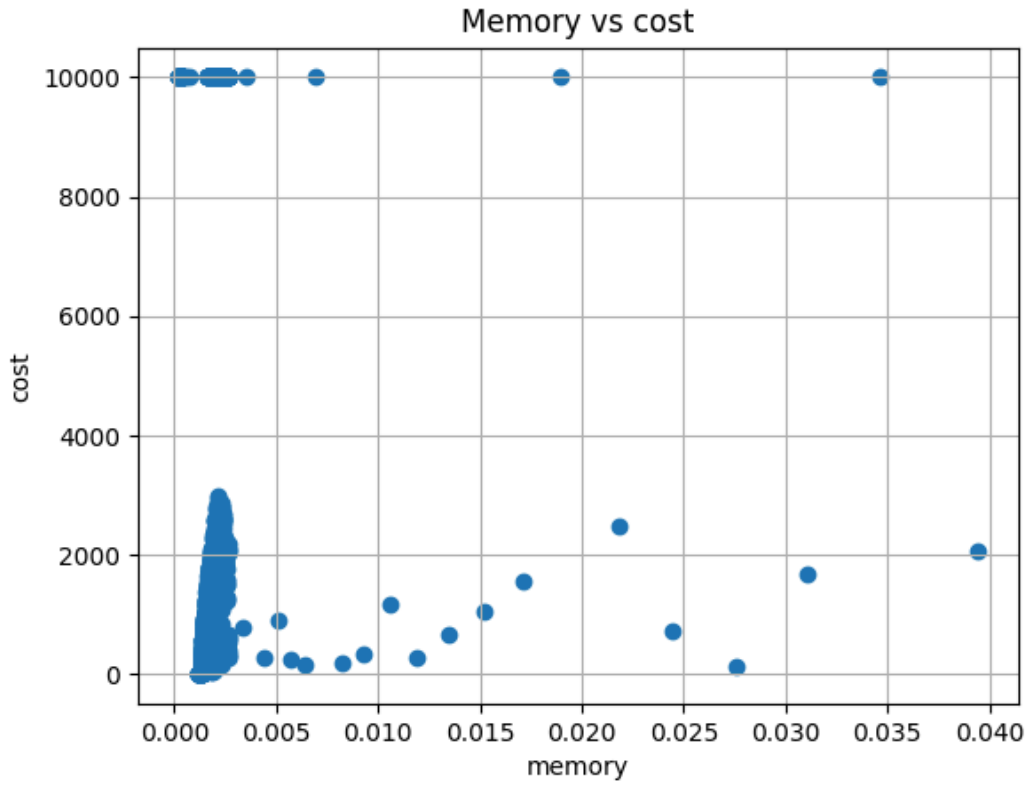
```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

PART F

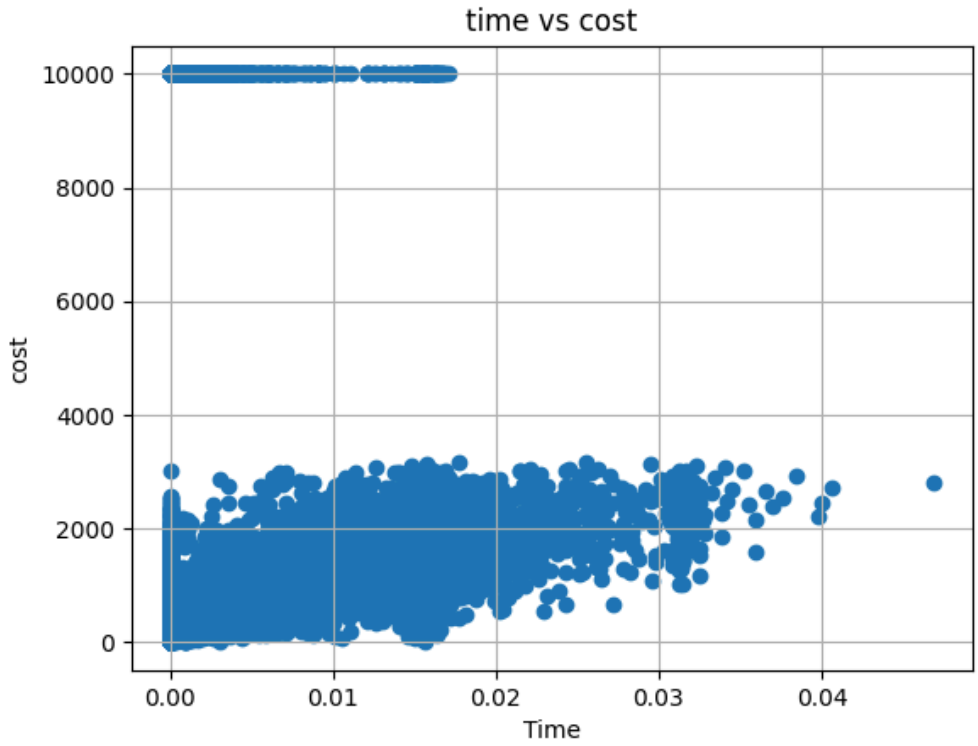
This section provides three scatter plots for each algorithm: 'Time vs Cost,' 'Memory vs Cost,' and 'Time vs Memory.' A brief analysis of the graphs follows this.

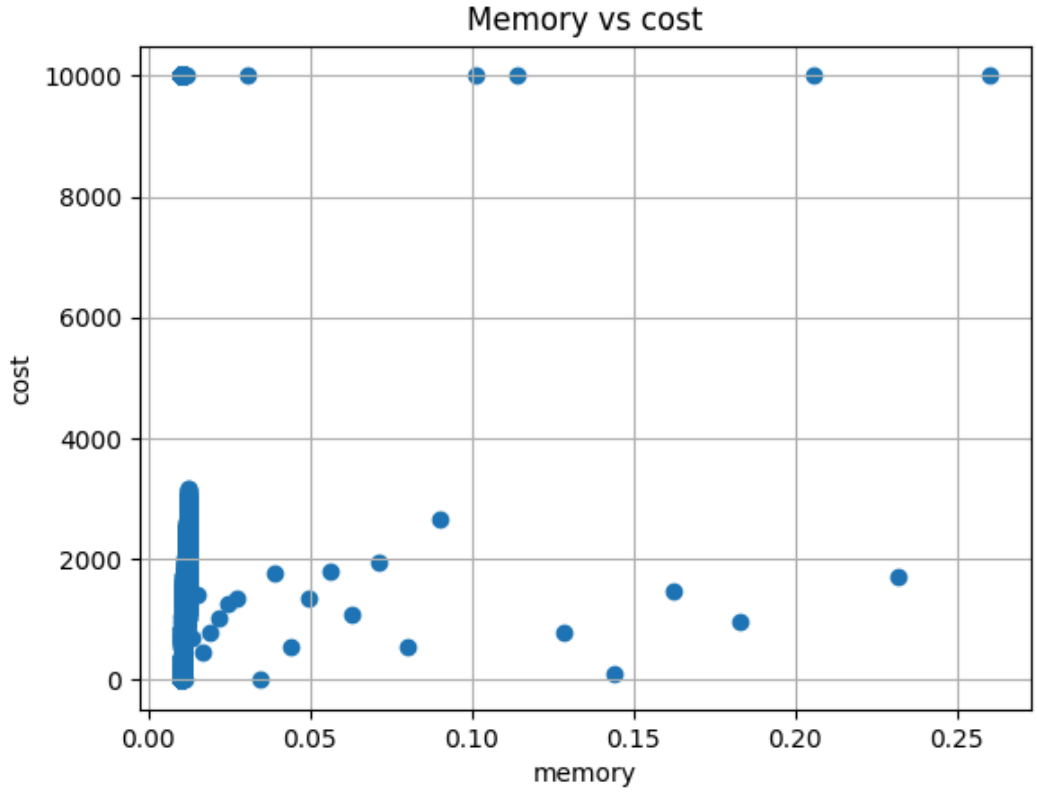
Iterative Depth Search Algorithm



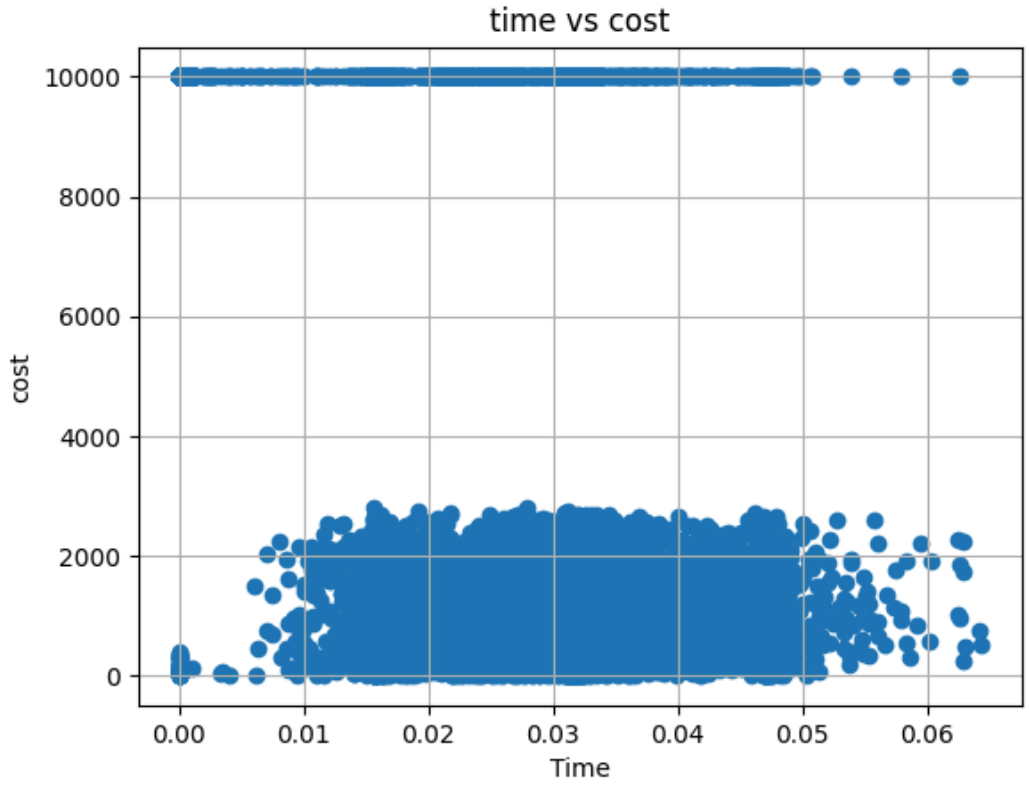


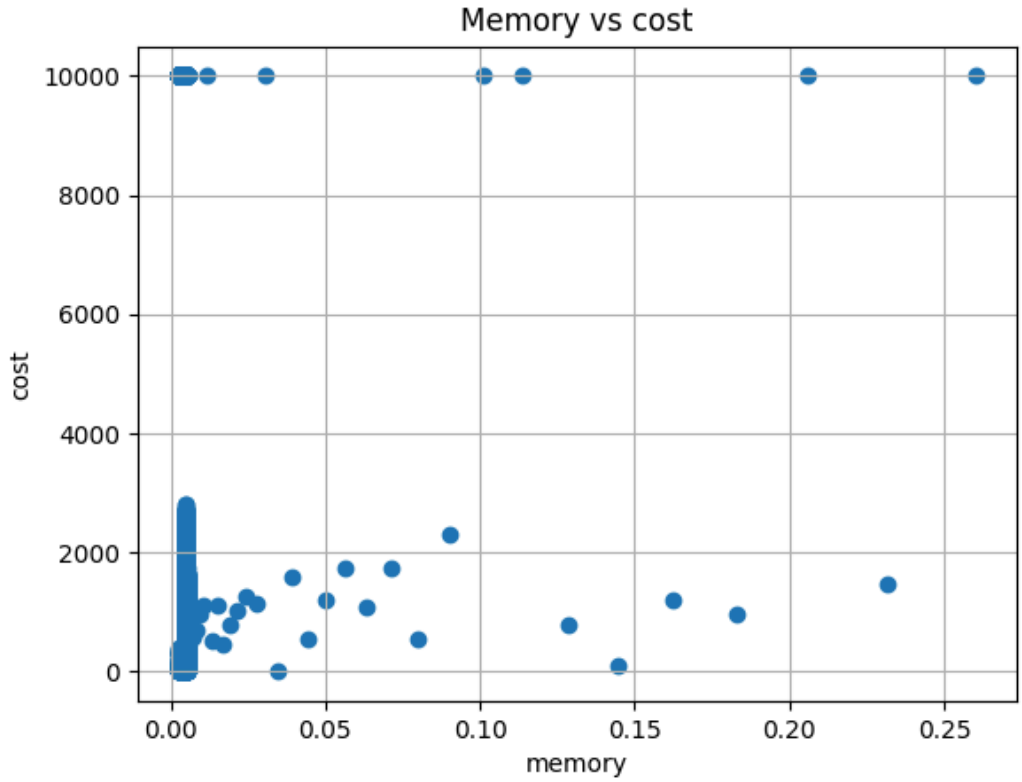
Bi-Directional Search Algorithm



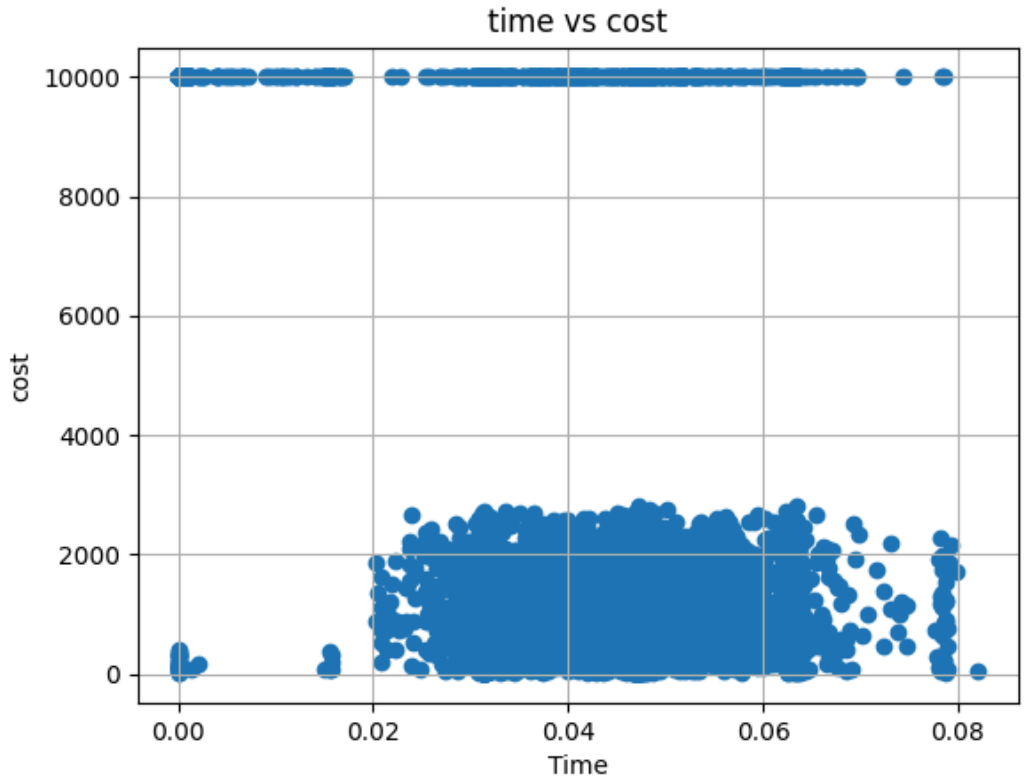


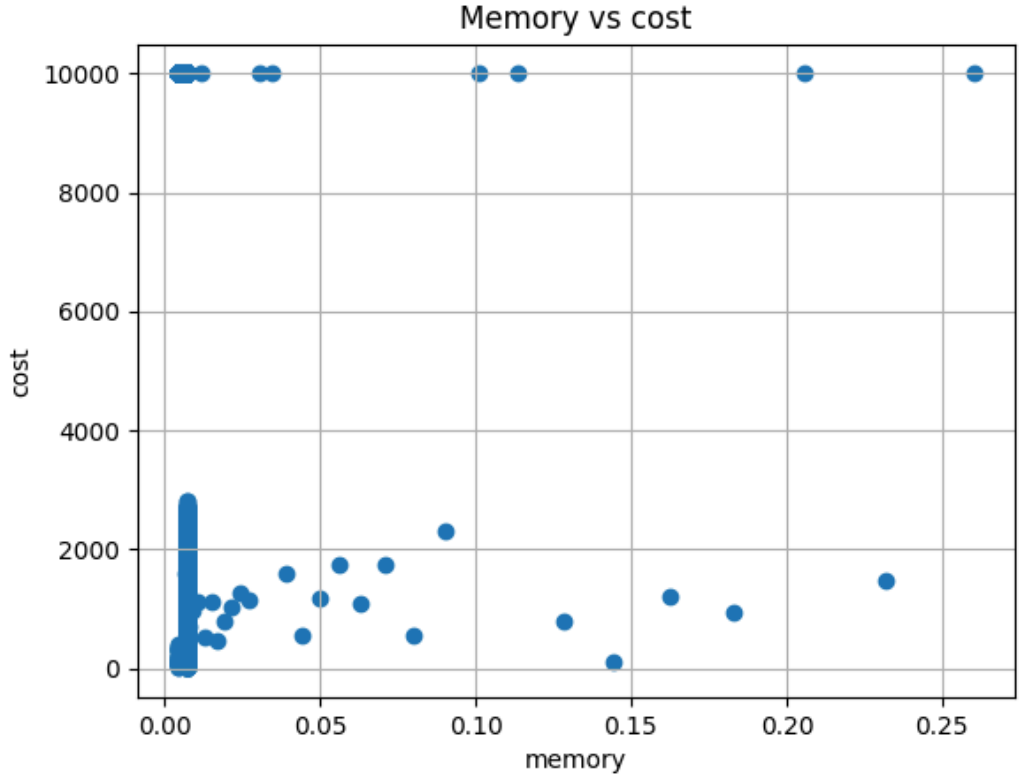
A* Search Algorithm





Bi-directional Heuristic Search Algorithm





NOTE: The scatter plot is obtained by running each algorithm for every pair of nodes present in the adjacency matrix. Hence, each point on the plot refers to a different pair *(start_node, goal_node)*. The number of pairs of nodes in the IDS algorithm was limited to 2500 due to computational and time constraints. Note that the cost for a “no solution” path is represented by 10,000 units, a sufficiently large number to represent the outliers.

Units for time axis: seconds

Units for memory axis: MiB

Units for cost: As defined by the problem statement (path distance).

Obtaining Performance Metrics:

- **Time:** The *time* library in python allows us to evaluate a function's start and end times. We used the difference in these values to calculate an approximation of the run time of the algorithm for each pair of nodes.
- **Memory:** The *tracemalloc* library evaluates the average and peak memory consumption for a given code segment. We considered the peak memory consumption in MiBs to calculate the worst-case space complexity of the algorithms for each pair of nodes.
- **Cost:** The cost is defined as the actual path length *g(n),* which is the sum of the path lengths for adjacent pairs of nodes in the output.

Inference:

- The cost values for heuristic search algorithms are lower than those of uninformed algorithms.
- The IDS algorithm's time consumption is exponentially high compared to other algorithms. It is so high that the time axis range is changed to seconds instead of milliseconds.
- The time consumption of the Bi-directional algorithms is lower than their uni-direction counterparts.
- Memory usage of uninformed algorithms is slightly better than heuristic search algorithms.

Conclusion

Advantages of Heuristic Algorithms:

Optimality: It is clear from the scatter plots that the cost values of A* and the Bi-directional Heuristic algorithm are lower than the uninformed ones. Thus, heuristic algorithms are guaranteed to find an optimal solution.

Efficiency: Heuristic algorithms reduce the search space and may lead to lower memory consumption. They follow a heuristic function, so instead of blindly searching, they reduce the search space by exploring the nodes that may optimize the solution.

Disadvantages of the Heuristic Algorithms:

Complexity: The algorithms are much more complex in this case. They also take more time than the Bi-BFS algorithm, which takes the shortest time out of the 4 provided algorithms. This is due to the fact that the bidirectional search algorithm only focuses on finding the first solution, that is, the path with the lowest depth. By discarding the regard for optimality, it can run in a much shorter time.

Infinite Search Space: This drawback of the given heuristic algorithm was not showcased due to the provided search space being finite. However, if the search space was spread infinitely or the number of nodes was very large, the IDS algorithm would be the best fit. Since it has a depth threshold, we can control our search space. In contrast, heuristic search algorithms without a depth threshold would keep running infinitely until the goal is reached. In this regard, the Iterative Depth First Search algorithm outperforms all the other provided algorithms.