CSE571 Artificial Intelligence Final Project: Bidirectional Search

Sai Likhith Yadav Seera 1224791490
Deep Zaveri 1230029231
Sanidhya Chauhan 1229529043
Varshini Rao Venkateshwara Rao 1229838274

***Abstract*—We have implemented the MM bidirectional search algorithm, as described in the paper "Bidirectional Search That is Guaranteed to Meet in the Middle." We have conducted tests and analysed the results within the Pacman domain. Our primary measure of algorithm efficiency is the number of search nodes expanded. We compared the performances between the bidirectional search algorithm and other conventional search algorithms: BFS, DFS, Uniform Cost, and A* search across various environments, problem specifications, task sizes, and complexities. The experimental results show the efficiency and effectiveness of each algorithm for path finding based on search nodes expanded and processing time. Our results demonstrate that MM has lower time complexity than the other algorithms. We then conducted a statistical analysis using a T-test to draw further conclusions from the results.**

***Index Terms*—MM Bidirectional Search, Depth First Search, Breadth First Search, Uniform Cost Search, A* Search, Heuristic Value.**

I. Introduction

We have encountered several widely used search algorithms, including BFS (Breadth First Search), DFS (Depth First Search), A* search, and Uniform Cost Search (UCS), each with its own advantages and drawbacks.

DFS explores the depths of the tree, expanding nodes sequentially, but it can be excessively time-consuming and may expand unnecessary nodes. BFS addresses the time constraint issue by expanding one full level of the tree before moving to the next level. However, it suffers from high space complexity due to the memory required for this process. UCS methodically explores every potential path to find the most optimal one, but it consumes a significant amount of space, as the priority queue must be constantly sorted to efficiently explore paths. The A* approach incorporates a heuristic to calculate cost but shares the time and space complexity disadvantages of other algorithms.

In our project, we have endeavored to implement another search algorithm, Bidirectional Search, which is designed to meet in the middle. This method involves searching from both the start and goal nodes simultaneously, with one search moving from the start node towards the goal and the other from the goal node towards the start. We halt the searches once they meet in the middle, thus minimizing the unnecessary expansion of nodes. We plan to test and compare the performance of Bidirectional Search with the aforementioned algorithms.

II. Technical Approach

The given paper [1] discusses a novel bi-directional heuristic search algorithm called MM, aiming to find optimal solutions efficiently in problem instances represented as state spaces with non-negative edge weights. Key definitions and terminologies are established, including problem instances defined as pairs of start and goal states (start, goal), and distances between states denoted as d(u, v). The primary objective of the search is to discover a least-cost path from the start state to the goal state, where C* represents the cost of an optimal solution.

The algorithm employs separate forward and backward searches, denoted by variables with subscripts F and B respectively. Each direction utilizes an admissible front-to-end heuristic. States are categorized based on their distance from the start and goal states, dividing the state space into six distinct regions. The regions include near, far and remote for both start and goal states. The algorithm prioritizes nodes on the Open list based on a novel approach, considering the maximum of f-value and twice the g-value of a node.

$$pr_F(n) = max(f_F(n), 2g_F(n)) \quad (1)$$

MM's unique approach to prioritizing nodes on the Open list, considering a combination of f-value and twice the g-value of a node, contributes to its efficiency in finding optimal solutions. Moreover, the algorithm's properties, including its ability to avoid expanding states that are far from the start or goal and its guarantee of optimality, reinforce its effectiveness in heuristic search tasks.

As mentioned in [1] MM stops when

$$U <= max(C, fmin_F, fmin_B, gmin_F + gmin_B + \epsilon) \quad (2)$$

where
fmin, gmin = minimum f-value and g-value in open set
ϵ = cost of the cheapest edge in the state-space

A. Properties of MM as mentioned in [2]

1) The forward and backward search of MM never expand the states which are far or remote from the start or the goal state respectively.
2) Nodes whose f-value are greater than C* are never expanded.
3) MM returns C*.

4) MM does not expand a path more than one time if there exists such a path from start to goal and the heuristics are consistent.

Additionally, MM0, a variant of MM with zero heuristic values, is introduced as a brute-force approach. Overall, MM and its variant MM0 offer efficient and optimal solutions for heuristic search problems.

Since, Bidirectional A* search simultaneously explores from both the start and goal nodes to find the optimal solution efficiently for a heuristic function, in this implementation we are using two heuristic functions 'manhattanDistance' and 'euclideanDistance' in Python.

The implementation consists of several functions:

- **flipActionPath:** Reverses a list of actions, used to reverse the actions from one end of the search to the other in bidirectional search.
- **manhattanHeuristic:** Computes the Manhattan distance heuristic for a given position and problem.
- **euclideanHeuristic:** Computes the Euclidean distance heuristic for a given position and problem.
- **graphTreeBiSearch:** Implements the bidirectional A* search algorithm, maintaining two priority queues for forward and backward searches. It expands nodes alternately from both ends until they meet in the middle.
- Heuristic-based search functions:
  1) **biDirectionalAStarManhattanHeuristicSearch:** Performs bidirectional A* search using the Manhattan distance heuristic.
  2) **biDirectionalAStarEuclideanHeuristicSearch:** Performs bidirectional A* search using the Euclidean distance heuristic.
  3) **biDirectionalAStarSearchBruteForce:** Performs bidirectional A* search without any heuristic (equivalent to uniform cost search).

The psudocode for the implementation is as follows:

Algorithm 1 Flip Action Path

function flipActionPath(actions): // Reverse the list of actions
reversed_actions ← []
for each action in reversed(actions) **do**
  add action to reversed_actions
end for
return reversed_actions

Algorithm 2 Manhattan Heuristic

function manhattanHeuristic(position, problem, info): // Compute Manhattan distance heuristic
xy1 ← position
xy2 ← problem.goal **if** 'flip' not in info **or** info['flip'] is False **else** problem.startState
return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

Algorithm 3 Euclidean Heuristic

function euclideanHeuristic(position, problem, info): // Compute Euclidean distance heuristic
xy1 ← position
xy2 ← problem.goal **if** 'flip' not in info **or** info['flip'] is False **else** problem.startState
return $((xy1[0] - xy2[0])^2 + (xy1[1] - xy2[1])^2)^{0.5}$

Algorithm 4 Graph Tree Bi-Search

function graphTreeBiSearch(problem, fringe_forward, fringe_backward, search_type, heuristic):
startingNodeF ← endingNodeB ← problem.getStartState()
endingNodeF ← startingNodeB ← problem.getGoalState()
initialize opensetForwards, closedsetForwards, opensetBackwards, closedsetBackwards
flipHeuristic ← {}
flipHeuristic['flip'] ← False
initialize startingNodeF in opensetForwards
push startingNodeF into fringe_forward
flipHeuristic['flip'] ← True
initialize startingNodeB in opensetBackwards
push startingNodeB into fringe_backward
U ← infinity
eps ← 1
while fringe_forward is not empty and fringe_backward is not empty **do**
  curMinNodeF ← peek from fringe_forward
  compute priorityMinForwards
  curMinNodeB ← peek from fringe_backward
  compute priorityMinBackwards
  compute C
  if U ≤ lowerCostBound or U ≤ C **then**
    return actionPath
  end if
  if C == priorityMinForwards **then**
    expandNodeForward()
  else
    expandNodeBackward()
  end if
end while
return actionPath

Algorithm 5 Bi-Directional A* Manhattan Heuristic Search

function biDirectionalAStarManhattanHeuristicSearch(problem, heuristic):
priority_queue_fwd ← PriorityQueue()
priority_queue_bwd ← PriorityQueue()
return graphTreeBiSearch(problem, priority_queue_fwd, priority_queue_bwd, 'MM_M', heuristic)

Algorithm 6 Bi-Directional A* Euclidean Heuristic Search

function biDirectionalAStarEuclideanHeuristicSearch(problem, heuristic):
priority_queue_fwd ← PriorityQueue()
priority_queue_bwd ← PriorityQueue()
return graphTreeBiSearch(problem, priority_queue_fwd, priority_queue_bwd, 'MM_E', heuristic)

Algorithm 7 Bi-Directional A* Search Brute Force

```
function biDirectionalAStarSearchBruteForce(problem, heuristic):
priority_queue_fwd ← PriorityQueue()
priority_queue_bwd ← PriorityQueue()
return graphTreeBiSearch(problem, priority_queue_fwd, priority_queue_bwd, 'MM0', heuristic)
```

III. Results, analysis and discussions

Table-1 evaluates the performance of different algorithms when applied to mazes categorized into Contours, Open, Small, and Medium types. The algorithms considered for comparison include Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), A* search algorithm, and three variants of Bidirectional search (Bi-Dir) with different heuristic functions (Manhattan, Euclidean, and 0). The evaluation metrics used include the cost of the path found, the number of nodes expanded during the search process, whether the algorithm successfully found a solution (Win?), and a calculated score reflecting the overall performance.

TABLE I
Comparision of the algorithms with different types of Contours, Open, Small and Medium Mazes

Maze	Algorithm	Cost	Nodes Expanded	Score
Contours	BFS	13	170	497
Contours	DFS	85	85	425
Contours	UCS	13	170	497
Contours	A*	13	49	497
Contours	Bi-Dir Manhattan	13	41	497
Contours	Bi-Dir Euclidean	13	46	497
Contours	Bi-Dir 0	13	90	497
Open	BFS	54	682	456
Open	DFS	298	576	212
Open	UCS	54	682	456
Open	A*	54	535	456
Open	Bi-Dir Manhattan	54	441	456
Open	Bi-Dir Euclidean	54	443	456
Open	Bi-Dir 0	54	443	456
Small	BFS	19	92	491
Small	DFS	49	59	461
Small	UCS	19	92	491
Small	A*	19	53	491
Small	Bi-Dir Manhattan	19	37	491
Small	Bi-Dir Euclidean	19	41	491
Small	Bi-Dir 0	19	50	491
Medium	BFS	68	269	442
Medium	DFS	130	146	380
Medium	UCS	68	269	442
Medium	A*	68	221	442
Medium	Bi-Dir Manhattan	68	183	442
Medium	Bi-Dir Euclidean	68	183	442
Medium	Bi-Dir 0	68	184	442

Table-II extends the comparison to mazes characterized as BigMaze1, to BigMaze15. Similar to Table 1, various algorithms are assessed for their effectiveness in solving these larger-scale maze problems. The evaluation metrics remain consistent, with considerations for cost, nodes expanded, successful solution finding, and overall performance score. Here, the results of the Big Maze problems are shown for five different instances. The remaining ones are present in the output file in the repository for further reference.

TABLE II
Comparision of the algorithms with different types of Big Mazes

Maze	Algorithm	Cost	Nodes Expanded	Win?	Score
BigMaze1	BFS	78	393	Yes	432
BigMaze1	DFS	78	565	Yes	432
BigMaze1	UCS	78	393	Yes	432
BigMaze1	A*	78	242	Yes	432
BigMaze1	Bi-Dir Manhattan	78	205	Yes	432
BigMaze1	Bi-Dir Euclidean	78	222	Yes	432
BigMaze1	Bi-Dir 0	78	225	Yes	432
BigMaze2	BFS	74	376	Yes	436
BigMaze2	DFS	214	402	Yes	296
BigMaze2	UCS	74	376	Yes	436
BigMaze2	A*	74	244	Yes	436
BigMaze2	Bi-Dir Manhattan	74	197	Yes	436
BigMaze2	Bi-Dir Euclidean	74	214	Yes	436
BigMaze2	Bi-Dir 0	74	216	Yes	436
BigMaze3	BFS	158	622	Yes	352
BigMaze3	DFS	186	459	Yes	324
BigMaze3	UCS	158	622	Yes	352
BigMaze3	A*	158	550	Yes	352
BigMaze3	Bi-Dir Manhattan	158	514	Yes	352
BigMaze3	Bi-Dir Euclidean	158	514	Yes	352
BigMaze3	Bi-Dir 0	158	514	Yes	352
BigMaze4	BFS	68	663	Yes	442
BigMaze4	DFS	110	465	Yes	400
BigMaze4	UCS	68	663	Yes	442
BigMaze4	A*	68	276	Yes	442
BigMaze4	Bi-Dir Manhattan	68	234	Yes	442
BigMaze4	Bi-Dir Euclidean	68	232	Yes	442
BigMaze4	Bi-Dir 0	68	239	Yes	442
BigMaze5	BFS	77	500	Yes	433
BigMaze5	DFS	77	422	Yes	433
BigMaze5	UCS	77	500	Yes	433
BigMaze5	A*	77	342	Yes	433
BigMaze5	Bi-Dir Manhattan	77	302	Yes	433
BigMaze5	Bi-Dir Euclidean	77	344	Yes	433
BigMaze5	Bi-Dir 0	77	339	Yes	433

```
(CSE571) C:\Users\likhi\OneDrive\Desktop\CSE-571 AI\Team Project\Search>python tTest.py
                                Algorithm  Cost p-value  Score p-value
0                                     BFS      1.000000       1.000000
1                                     DFS      0.023943       0.023943
2                                     UCS      1.000000       1.000000
3                                      A*      1.000000       1.000000
4  Bi-Directional with Euclidean Heuristic      1.000000       1.000000
5  Bi-Directional with Manhattan Heuristic      1.000000       1.000000
6  Bi-directional with Euclidean Heuristic      1.000000       1.000000
```

Fig. 1. Results of t-Test

IV. Conclusions and discussions

We implemented MM using Manhattan distance, MM using Euclidean distance and MM0 with null heuristics. We compared the results of this algorithm with uninformed search algorithms like BFS and DFS, providing the smallest number of expanded nodes. In informed search algorithm like A* and UCS, various heuristic functions were analyzed and performed in Pacman domain. The best performance was obtained using MM algorithm and it outperformed A* and other algorithms on various levels of domains (size level) and heuristic functions.

References

[1] R. Holte, A. Felner, G. Sharon, and N. Sturtevant, "Bidirectional Search That Is Guaranteed to Meet in the Middle", AAAI, vol. 30, no. 1, Mar. 2016.

[2] Holte, R. C.; Felner, A.; Sharon, G.; and Sturtevant, N. R. 2015. Bidirectional search that is guaranteed to meet in the middle: Extended Version. Technical Report TR15-01, Computing Science Department, University of Alberta.
[3] Pacman Project adapted from Berkeley CS188