Implementation of Bidirectional search in Pacman Domain

Archit Bansal, Mitchell Bucklew, Michael Tompkins

***Abstract*— In this paper we explore bidirectional search, as implemented in "Bidirectional Search That Is Guaranteed to Meet in the Middle" in the Pacman domain. We present experiments with bidirectional search to compare them to other search algorithms in Pacman layouts. Finally, we draw a conclusion based on the experimental results.**

***Index Terms*—Bidirectional Search, Pacman Domain, Depth First Search**

I. Introduction

There are many search algorithms that contain unique benefits and drawbacks. Depth-first search (DFS) is an algorithm that explores nodes immediately upon discovery. Breadth-first search (BFS) is a similar algorithm but explores all nodes of the same depth sequentially. One of the issues with bread first search however is the excessive memory usage of the algorithm. Instead of BFS, iterative deepening with depth-first search can be used which uses much less memory. This algorithm works by running an otherwise normal DFS algorithm with a depth limit starting at 0 and sequentially increasing the limit. Depth limited DFS of 1 is identical to BFS in theory and performance.

This technique can be utilized in unique ways. Say we have a relatively simple search problem, where we are searching for a fixed goal. Using traditional BFS and DFS algorithms memory and resource usage can grow exponentially in certain cases. In DFS, open mazes perform poorly and are unlikely to find the optimal solution. BFS will find the optimal solution but visit many nodes in these open worlds. We will see algorithm performance depends on the scenario, and in further testing, we will consider the average performance over all available scenarios in the Pacman domain.

In certain situations, BFS and DFS can be improved by using a search strategy. In more complicated environments, a search strategy or search heuristic is needed. Especially in non-deterministic environments where decisions are being provided immediately, such as environments controlling Mr. Pacman in real-time against ghosts. This is implemented as A* search. Using a search strategy can help to reduce the number of nodes visited.

Bidirectional search is yet another search algorithm, where not only is the goal location searched for from the beginning, but the start is searched from the goal until the two searches meet in the middle. This technique can be used with BFS search to search blindly with BFS or with a search strategy like A*.

We present both a bidirectional heuristic (Bi-HS) and bidirectional brute-force search (Bi-BS) in the Pacman domain. These algorithms are implemented as presented in "Bidirectional Search That Is Guaranteed to Meet in the Middle". We have designed experiments to test these algorithms and rank them next to other various search techniques in the Pacman domain. We present a statistical analysis of the number of nodes expanded. A t-test is performed on the number of nodes to determine the statistical significance. By analyzing the average amount of node expanded overall maze instances in the Pacman domain we can conclude the most efficient search algorithm

II. Technical approach

We implemented this Bidirectional search as described in "Bidirectional Search That Is Guaranteed to Meet in the Middle". We have applied a search strategy technique to Bidirectional search by using a heuristic. Additionally, we have implemented a Queue based Bidirectional search labeled a bd0. This is also known as brute force Bidirectional search. We utilized the Food Search problem from the Pacman domain. This is a search problem where Pacman navigates a maze to find the single food in the world. As is the case with the large maze in figure 1.

Normally in A* search, the goal state is detected when the goal is expanded using the problem.isGoalState(node) function. This function does not work in our case as either search will never expand their goal node, granted they meet in the middle as intended. To complete this task, we modified the isGoalState function of the food search problem. This function now takes two arguments, the current state and the visited nodes of the other corresponding search.

Modified goal Test

```
def isGoalState(self, state, other_visited=None):
    if other_visited is None:
        isGoal = state == self.goal
    else:
        isGoal = state in other_visited
```

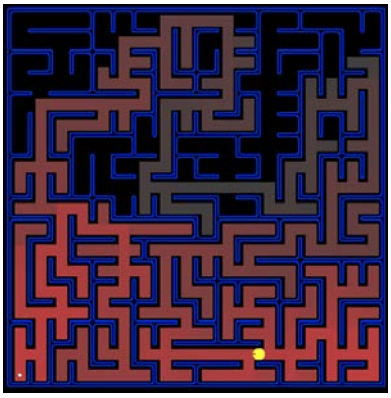

Figure 1: A food search problem on bigMaze

There are two cases in this goal test function. The first in the if statement is for Non-Bidirectional search functions to ensure compatibility with other algorithms like A*, BFS, DFS. The second case contained in the else statement is for our bidirectional search. The node is checked to have been visited by the other search algorithm, and if it has the goal has been reached and the two algorithms have "met in the middle".

There are two variations to the bidirectional search. One with a search strategy (MM) and one without a search strategy (MM0). Both use BFS and A* respectively. The difference is the data structure which contains the frontier, which contains the nodes to be visited next. MM0 or brute force bidirectional search uses a Queue data structure with a first-in, first-out policy. MM0 is like BFS. MM utilizes a min priority queue, which gives easy access to the node with the lowest priority. The priority being the A* cost metric below:

A* Cost Function
problem.getCostOfActions() = g(n) *heuristic(state, problem) = f(n)* *f(n)= h(n) + g(n)*

The heuristic we are utilizing, in this case, is an implementation of the Manhattan distance heuristic. There are two cases for each search. For the search starting from the beginning, the heuristic is the Manhattan distance to the goal node. For the search starting from the end, the heuristic is the Manhattan distance to the start node.

We present the bidirectional search pseudocode in the Pacman world. The only difference between this implementation and brute force bidirectional search is the data structure of the frontier and the fact a heuristic is used. revDirections is a function which reverses the order of a list of directions and translates all moves into their opposite move respectively. For example, the directions, "South, East, East" will be transformed into "West, West, North". This is useful for reversing the path of the search from the goal to the start, as the order of these directions will be reversed from the actual solution. Additionally, visited dictionaries contain paths to each node throughout each search.

```
Bidirectional Search
   q1, q2 = PriorityQueue(startstate), PriorityQueue(goal)
   vis1,vi2 = {}. {}
   while q1 and q2 not empty:
      pos = q1.pop()
      if isGoalState(pos):
         return path + revDirections( vis2[pos] )
      for state in successors( pos ):
         if state is vis1:
            continue
         q1.push(state , f(state))
         vis1[state] = path + nextMove

      pos = q2.pop()
      if isGoalState(pos):
         return path + revDirections( vis2[pos] )
      for state in successors( pos ):
         if state is vis1:
            continue
         q1.push(state , f(state))
         vis1[state] = path + nextMove
return []
```

III. Results

3.1 Comparison of the Number of Nodes Expanded

TABLE I
NUMBER OF NODES EXPANDED

Maze	BFS	DFS	UCS	A*	Bi Direction	Bi Direction 0
contours	167	85	170	49	45	170
small	91	59	92	53	38	92
medium	268	146	269	221	142	269
big	620	390	620	549	495	620
open	682	806	682	535	108	682
Avg	365.6	297.2	366.6	281.4	165.6	366.6

This table shows the number of nodes expanded for each maze respectively. Here we compare A* and Bidirectional search used with a Manhattan distance heuristic.

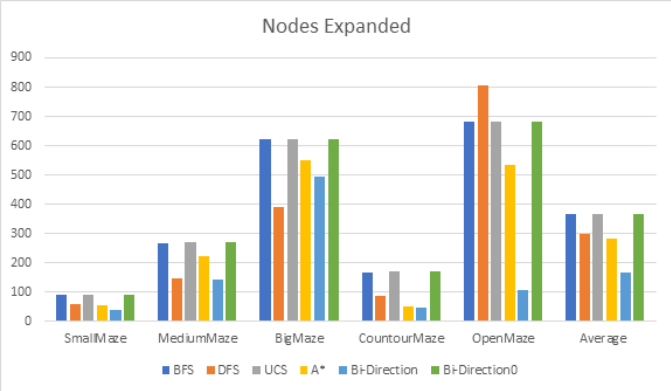

Figure 2: All nodes expanded plotted on a bar graph

In Figure 2, we are comparing the amounts of node expanded between the various algorithms DFS, BFS, UCS, A*, Bi-directional, and Bi-directional 0. This graph was made using the table provided above. Now let's start with analysis. First, we have the small maze in which we see that the three algorithms i.e. BFS, UCS, and bi-directional 0 are performing similarly and have the highest node expansion. Then we have DFS and A* having somewhat the same results and node expanded are still less than the other BFS, UCS, and bidirectional 0. Then we have the bi-directional search which has the least amount of node expanded at 38.

Next, we consider the medium maze. We again see that the highest node expansion is for BFS, UCS, and bi-directional 0. Coming in at second, we have the A* algorithm with the count of search nodes at 221. DFS and the bidirectional have almost the same node expansion but still bidirectional has the least number of nodes expanded with 142 as compared to DFS which has 142 nodes expanded.

Then we have the big maze in which the results are the same where the highest node expanded belongs to BFS, UCS, and bidirectional 0. We see a critical difference here as our bidirectional search algorithm does not have the least number of nodes expanded as at 390 nodes DFS has the least count. However, the Bi-directional search is still better than the A* in all the mazes above.

Now we have the contours maze here we can the similar trend the highest node expanded belong to UCS, BFS, and bi-directional 0. Then we see that DFS has the second highest node expanded and then this time we see that the A* algorithm and the bidirectional search have almost the same node expanded but bidirectional still has the least node expanded at 45.

Then our last maze is the open maze. This is one maze where we can clearly see that the bi-directional search has destroyed the other algorithm in terms of node expansion. BFS, A* bidirectional have the same amount of node expanded and our second highest in terms of node expanded and then we have the DFS which has performed the worst of all the algorithms with the node expansion of 806 compared to bidirectional which has only 108 nodes expanded. Hence, we can say that the bidirectional search has outperformed almost every time.

3.2 Comparison of the cost of paths found

TABLE 2
COST OF PATHS FOUND

Maze	BFS	DFS	UCS	A*	Bi Direction	Bi Direction 0
contours	13	85	13	13	13	13
open	54	298	54	54	60	54
Medium	68	130	68	68	68	68
Avg	45	171	45	45	47	45

This table shows the cost of the path found for each maze respectively. Here we compare A* and Bidirectional search used with a Manhattan distance heuristic. Here we only compare mazes with multiple routes from the beginning to the end.

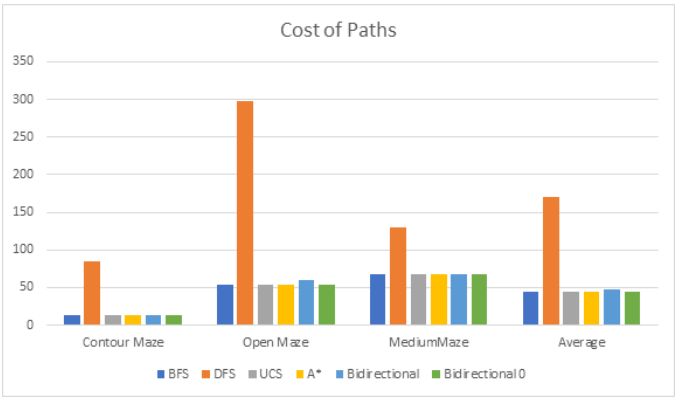

Figure 3: Costs of path found plotted on a bar graph

In all the graphs we can see that the cost of DFS has always been higher than all the algorithms. And the other algorithms have performed similarly. We can conclude that DFS performed poorly because it had the highest path of cost.

3.3 Comparing t- test value for each algorithm against bidirectional search

TABLE 3
P VALUES FOR EACH ALGORITHM

BFS	DFS	UCS	A*	Bi Direction 0
0.024372	0.156675	0.023802	0.056195	0.023802

We are comparing -test values between all the algorithms and bidirectional search. We are taking the confidence value around 95% and hence predicting the result that any value greater than .05 will be insignificant to compare and which tells

us that the algorithm must have close values with the bidirectional search. And any value less than or equal to 0.05 will mean that the bidirectional search is way better than that algorithm and the algorithm must have a significant difference in values.

The minimum values were obtained from the UCS and Bidirectional 0 each of them has the same value of .023802 and hence we can say that bidirectional search is performing better than these algorithms since their values are less than .05. Also, this means that the value of the search node expanded between these two algorithms and bidirectional has a significant difference. We can also say by looking at the data that UCS and Bidirectional 0 might have the same type of performance since they gave the same values. Then at second, we see that BFS too has a lower value of .024 and is less than .05 and hence implies that the bidirectional search is better than BFS and their search nodes expanded to have a significant difference. A* algorithm has .056195 hence showing that it is better than UCS, BFS, DFS but still not better than the bidirectional search. Now with the value of 0.156675 DFS has the greatest t-test value and this means that the comparison between DFS and bidirectional is not that significant and these both algorithm must have outperformed each other at some point and hence we can say that DFS and bidirectional might have closer values than other algorithms.

IV. Conclusion

The results obtained point to the conclusion that for the given mazes and layouts Bi-Directional search (MM) outperforms all of the other algorithms. Note that this use case is fairly simple, and the results will most likely change in more complex use cases, as an example of this, search tasks with large branching factors will likely result in bidirectional searches performing on average worse due to there being a large potential for each direction of the search to branch into different locations, meaning they might not meet at an optimal midpoint, but would each require less space compared to a traditional A* approach. In addition, this search algorithm is only useful in cases where we already know the goal state, because Bi-Directional algorithms must start from the goal as well. Overall, there's a lot of research still to be done in this area, so expect more improvements in the future.

A. References

[1] Robert C. Holte, Ariel Felner, Guni Sharon, Nathan R. Sturtevant, "Bidirectional Search That Is Guaranteed to Meet in the Middle" Available: https://people.engr.tamu.edu/guni/Papers/AAAI16-MM.pdf, Accessed on: Dec. 7, 2020