Implementation and Analysis of Bi-Directional Search in the Pacman Domain

Luna, Juan, Meza, Edward, Ortiz Rivas, Juan, and Shankar, Arun

***Abstract*—In [1], Holte et al present a variation of bi-directional search, a search algorithm where a forward to the goal and backward to the start search are both conducted. Their variation is designed to "meet in the middle" or never have either half of the search expand a node further than the midpoint. Using the Pacman domain from our previous projects, we implemented bi-directional search in Python. Then, we compared it to already established search algorithms depth first search, breadth first search and A* search in terms of nodes expanded and total path cost. We did this by running all algorithms through nineteen different variations of the Pacman maze. Variations ranged in size and complexity. Our results show that in most cases bidirectional search outperformed the other algorithms in terms of nodes expanded both with and without a heuristic corroborating the claims in the paper. When exceptions occur, they lead to a higher path cost or can be attributed to an accurate heuristic.**

I. Introduction

Search can be broadly defined as trying to find a path from a starting node to a defined goal in a graph. Uninformed search, where the algorithm has no knowledge of the problem outside of its definition, includes breadth-first search (BFS), depth-first search (DFS), and their variations. Informed search, where the algorithm uses a heuristic to estimate before searching includes A* search and its variations. Each one of these algorithms has its strengths and weaknesses and no one algorithm is optimal in all scenarios. Thus, the search for a perfect search algorithm continues. In this paper, we will investigate a new type of algorithm, bi-directional search. Specifically, we will investigate a variation of bi-directional search designed to meet in the middle. This will be explained further in Section 2.

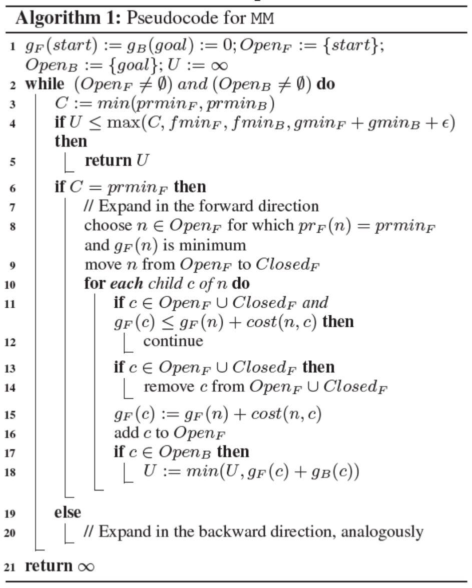
The rest of this paper will be organized as follows. Section 2 will explain the specific bi-directional search algorithm we will use in this project. Section 3 will present a detailed analysis of our bi-directional search implementation in the Pacman project domain as well as a comparison with BFS, DFS, and A* search in the same domain. Section 4 will detail the conclusions reached and section 5 will be a brief explanation of our team's dynamic.

II. Technical Approach

A bidirectional search is what it sounds like: conducting a search in two directions. One search goes forward from the start state to the goal and the other goes backward from the goal to the start state. A component of the theoretical approach to bidirectional search is that neither the forward nor backward searches expand a node where the g value (distance from the current state to the start or goal pending forward or backward search) is greater than C*/2 where C* is the optimal path cost for the whole graph. In other words, the forward and backward searches only search halfway and "meet in the middle." However, most bidirectional algorithms are not guaranteed to always meet in the middle. The one we are implementing though does.

The bidirectional search presented by Holte et al. is nicknamed MM, for meet in the middle. The forward and backward searches are extremely similar. For each node, there is a g value, h value (heuristic value defined the same way for the forward and backward searches), and an f value calculated as (g + h). There are also lists of open nodes (ones that need to be expanded) and closed nodes (ones already expanded) for both types of search. The ordering of nodes in the open list is determined by a priority of max (f, 2g) again defined the same way forward and backward. The node with minimum priority is placed at the top of each independent list and the overall node chosen to be expanded is the node that has the minimum priority between the two open lists.

The minimum priority goes first in each independent list and the overall node chosen to be expanded is the minimum of the first node in each list. The pseudocode is as follows:



Algorithm 1: Pseudocode for MM

```
$g_F(start) := g_B(goal) := 0; Open_F := \{start\};$
$Open_B := \{goal\}; U := \infty$
while $(Open_F \neq \emptyset)$ and $(Open_B \neq \emptyset)$ do
    $C := min(prmin_F, prmin_B)$
    if $U \leq \max(C, fmin_F, fmin_B, gmin_F + gmin_B + \epsilon)$
    then
        return $U$
    if $C = prmin_F$ then
        // Expand in the forward direction
        choose $n \in Open_F$ for which $pr_F(n) = prmin_F$
        and $g_F(n)$ is minimum
        move $n$ from $Open_F$ to $Closed_F$
        for each child $c$ of $n$ do
            if $c \in Open_F \cup Closed_F$ and
            $g_F(c) \leq g_F(n) + cost(n, c)$ then
                continue
            if $c \in Open_F \cup Closed_F$ then
                remove $c$ from $Open_F \cup Closed_F$
            $g_F(c) := g_F(n) + cost(n, c)$
            add $c$ to $Open_F$
            if $c \in Open_B$ then
                $U := min(U, g_F(c) + g_B(c))$
    else
        // Expand in the backward direction, analogously
return $\infty$
```

Figure 1: Psuedocode for Holte, et al's Algorithm [1]

Initially, the g values are 0 and only the start and goal nodes are added to the open lists. U is the minimal cost of the path: initially infinite. Then as long as both lists are not empty, we pick the minimum priority node in either list. If the cost is already at a minimum and cannot be lowered anymore by searching, we just return the path cost along with the list of actions that will take the agent to the goal. If not, and the node was expanded from the forward list, we do a typical A* like search. If the node was expanded from the backward list, we do an A* like search from the goal to the start state [1].

In the Pacman domain, we implemented a bidirectional search as an additional option in the search.py function using all the same command-line options as with DFS, BFS, and A*. The use of priorities in the algorithm seemed to indicate that priority queues would be the ideal data structure to implement the MM algorithm. We started by using the priority queues that were already implemented for previous Pacman projects. However, it immediately became apparent that this would not work. The algorithm required us to keep track of many things, such as the overall minimum g and f values and whether or not a specific node was present in either open list. We tried implementing helper functions that would solve these problems but the restrictions placed by a queue in that only the top element is visible were too great. We then decided to use plain lists. Although this meant each open list would be traversed many times to get the information needed thus having an impact on computation time it also gave us the flexibility necessary to implement the desired functionality.

Helper functions were created to retrieve a specific node and all its information, determine if a certain state is in an open list, remove a specified node from a given list and get the requisite minimums for g, f, and priority. Since backward search returned an action list that would create illegal actions if used normally, a function to mirror the actions and reverse their order was also implemented. The heuristic utilizes the “front to end” heuristic as noted in the paper by way of the Manhattan distance. A variation of the Manhattan heuristic was also implemented so that the agent searched for the goal rather than the start state. There were some bugs in which the algorithm iterated one more time than necessary which resulted in illegal actions being created. However this was easily fixed by checking the creation of such actions and preventing the algorithm from updating the action list. The rest of our code stems from the pseudocode.

III. Results

In this section, we performed the analysis of various search algorithms, namely BFS (Breadth-First Search), DFS (Depth-First Search), A* Search, A*H (A* with Heuristic), MM$_0$ (Brute Force Bi-Directional Search) and, MM (BDS with Heuristic) by running the algorithm through nineteen different layouts of the Pacman-Domain. We were able to perform a relative comparison between the Nodes expanded and Path Cost for each of the search algorithms per layout. However, to show a more accurate comparison between the algorithms, we plotted the normalized performance that is determined by the following formulae:

$$Node\ Performance = \left(\frac{Nodes\ Expanded\ by\ Algorithm}{Max\ Nodes\ Expanded\ by\ Layout}\right)^{-1}$$

$$Path\ Cost = \left(\frac{Path\ Cost\ by\ Algorithm}{Max\ Path\ Cost\ by\ Layout}\right)^{-1}$$

The findings and analysis of our results are given below.

A. *Comparison between MM$_0$ and BFS*

We compare the MM$_0$ vs. BFS algorithm and observe the node performances and infer that on an average case, the nodes expanded by the MM$_0$ is fewer than that of BFS and hence the node performance of MM$_0$ is better, but there are different scenarios in the Pacman Domain that need to be taken into consideration.

We observe that across the 19 layouts in Pacman, MM$_0$ outperforms BFS with the node performance. The only exception that we observe is in the layout: *trappedClassic* where BFS expands fewer nodes than MM$_0$ for the same path cost. This is the case where the Pacman agent is bound to die without completing the game.

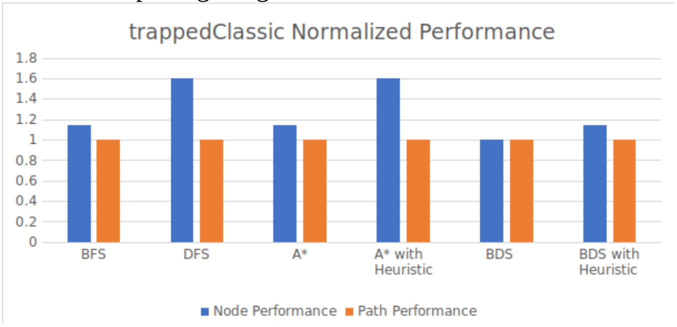


Figure 2: trappedClassic Normalized Performance

B. *Comparison between MM$_0$ and A**

In the near far region [1], A* expands many nodes. The nodes will have $g_F(n) <= C^*/2$ and A* prunes only the $h_F(n) > C^*/2$. For MM$_0$ nodes will not expand with $g_F(n) = C^*/2$, so we already have half the nodes expanded in MM$_0$ as compared to A*. In a near near region [1], MM$_0$ may expand zero nodes. As expected, A* expands lesser nodes than MM$_0$ in the far near region [1].

When we consider the example of the *smallMaze* layout we see that MM$_0$ expands only 49 nodes as compared to A* with 92 nodes expanded, which betters down when used with a heuristic when it still expands to 53 nodes. Hence, the performance for MM$_0$ is better.

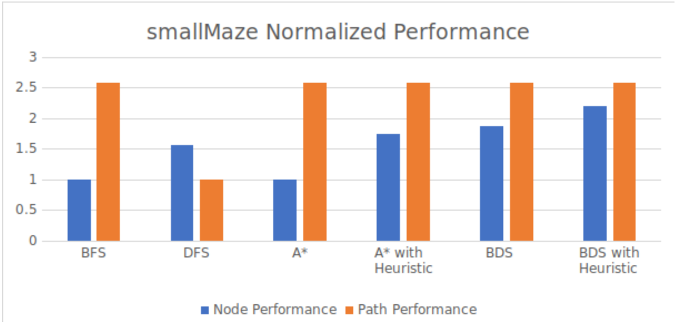


Figure 3: smallMaze Normalized Performance

There are however, exceptions to this scenario as well. We see that in some of the layouts, with the introduction of Manhattan heuristic, A* starts to significantly beat MM_0.

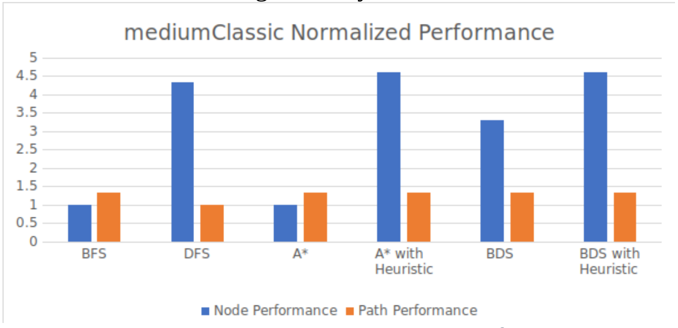


Figure 4: mediumClassic Normalized Performance

In the case above, nodes expanded with A* is 69 but with the introduction of the Manhattan Heuristic, it drops to 15, bettering the MM_0 algorithm with 21 nodes expanded. The reason this may happen is if the A* heuristic is extremely accurate, (i.e. Manhattan Distance). Hence, for the layouts where this distance is calculated correctly, this can outperform MM_0.

*C. Comparison between MM and A**

In this scenario, we introduce the heuristic into the Bi-Directional Search algorithm and we denote it as MM. Here we define B as h_B or the heuristic for MM backward search. This is the part described as the part of the search that is not pruned by A*. By definition, near far unpruned in MM is smaller than or equal to near far unpruned in MM_0, and near near unpruned in MM is smaller than or equal to near near in MM_0. Hence, MM is more advantageous over A* in the near near and near far region [1]. The practical implementation of which we see in the Pacman Domain layout analysis.

In at most 17 out of 19 layouts of the Pacman Domain, the MM algorithm outperforms the A* algorithm with heuristic taken as Manhattan distance. Taking the example of the *openMaze* layout in Pacman, the path performance remains the same for both A* and MM algorithms. But, when we analyze the total number of node explored, we observe that MM explores 245 nodes as compared to A* with heuristic which explores 535 nodes, which is more than two times that of MM. This is where we get to observe the true potential for the Bi-Directional search algorithm. We display our analysis as given below, through a comparative chart.

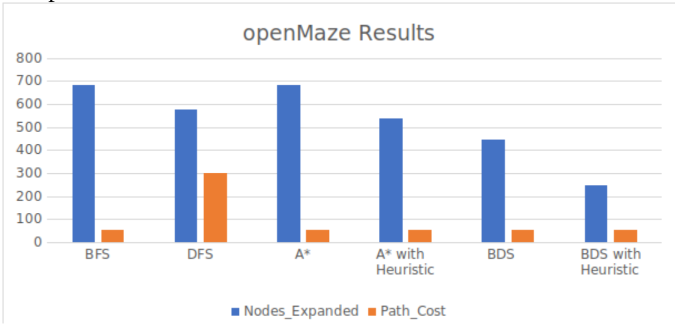


Figure 5: Comparison in Nodes Explored

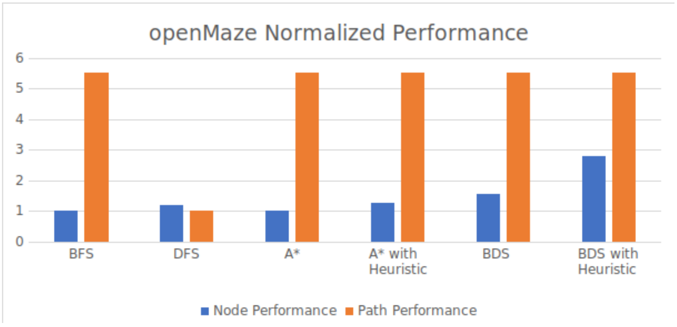


Figure 6: Comparison of Path Performance and Node Performance

We do have an exception in one of the Pacman layouts *bigMaze* where the A* with Manhattan heuristic performs slightly better than the MM algorithm. This is however possible when you take into consideration the far near in MM_0 region. The far near unpruned in MM_0 is smaller than backward far near in MM in this case due to the forward search that is involved. In our scenario, the nodes in the forward search $g_F(n) > C^*/2$ and heuristic function h_F estimates at most $C^*/2$. Whereas, the backward function $g_B(n) <= C^*/2$ and h_B needs to calculate the distance in order to prune it. That is why A* outperforms MM. The performance chart is shown below.

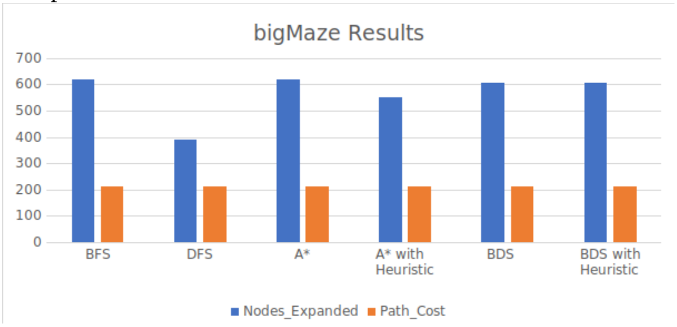


Figure 7: bigMaze Node Analysis

The A* with Heuristic has 549 nodes explored, does slightly better than MM which has 605 nodes explored even though the path cost remains the same.

D. Overall Takeaway

In this section, we infer our findings and discuss the overall analysis. After having implemented different search algorithms namely BFS, DFS, A*, A* with Heuristic, MM_0, and MM algorithms over 19 layouts to the Pacman Domain, we can verify that the average performance of MM (Bi-Directional Search) algorithm performed better than all the others. Although, the average normalized path cost performance was at ≈2.67, the major take away comes at the normalized performance of nodes explored for MM which is at 3.206 which is higher than any other algorithms on the list.

Algorithm	**Performance of Nodes Explored (Normalized)**	**Performance of Path Cost (Normalized)**
BFS	1.076	2.676
DFS	2.531	1.000

A*	1.076	2.676
A* H	2.777	2.676
MM_0	2.082	2.676
MM	3.206	2.672

Table 1: Normalized Average Performance

Another trend that we experience in our analysis with our search algorithms is that the inclusion of heuristic naturally helps the search agent in Pacman find a solution that may not be optimal but an acceptable solution. In 12 out of the 19 layouts, we noticed that MM algorithm performed better than MM_0 algorithm when comparing the path cost performance or the node explored performance. In 6 out of the remaining 7 cases, the heuristic function did not have an effect at all, the MM_0 and MM algorithm performed similarly. In one case that can be coined as an exception, the *tinyMaze* layout, the MM_0 performed slightly better than the MM algorithm. With identical path costs, the MM_0 explored only 11 nodes as compared to MM that explored 13 nodes as shown in the chart below.

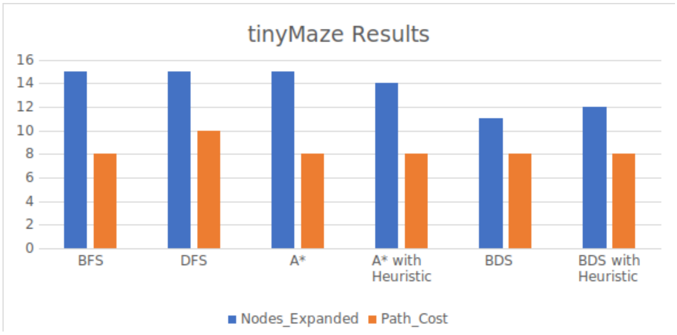


Figure 8: Inaccurate Heuristic Exception

The above analysis hence infers that a heuristic function may not always provide an optimal solution, and can be ineffective if an inaccurate heuristic-based search is applied to a scenario. Another takeaway from the observation is that an inaccurate heuristic can lead to a negative impact and cause more nodes to be expanded than it would in a search algorithm without a heuristic, as we observed in the *tinyMaze* layout.

IV. Team Effectiveness

Every team member contributed equally as follow:

- Juan Luna: Results and Analysis
- Edward Meza: Abstract, Introduction and Code Implementation
- Juan Ortiz Rivas: Conclusion and Code Implementation
- Arun Shankar: Results and Analysis

No problems were had and even in a virtual environment we were able to finish our paper.

V. References

[1] Holte, Robert C. et al. (2016, March). Bidirectional Search That is Guaranteed to Meet in the Middle. Presented at AAAI 2016.