Bi-directional search

Advaith Alenkrith Pagidipally, Mohammad Saad Mujeeb, Jaswanth Gorthi, Ahmed Shabbir

***Abstract*—This document discusses bi-directional search and its implementation(MM0 and MM algorithm). Search algorithms are important and used extensively in our everyday life. We find its implementation in various fields and applications such as searching in databases, GPS applications, and Recommendation systems. There are many important and useful searches such as D.F.S, B.F.S, binary search and others which we are going to compare against our MM0 and MM implementation. If we discuss the important parameters which determine the effectiveness of an algorithm, time complexity and number of nodes expanded stands out. In most of the search algorithms we start searching from an initial node(start state) and try to find the end state(goal state) by expanding a series of states in between them. These are called unidirectional searches as we are starting from the initial state and are driven to reach the end state. Although uni-directional searches find the solution it would take much less time for the bi-directional searches to achieve the same. In this document we will discuss in depth on how bi-directional search works and the implementation of two bi-directional algorithms MM0 and MM. After discussing the implementation we will discuss the results such as the cost, number of nodes expanded and score. Then we will evaluate t and anova tests and discuss the interpretations from the results.**

***Index Terms*—Bi-directional Search, MM0 algorithm, MM algorithm, t-test**

I. Introduction

BI DIRECTIONAL searches are the type of searches where we perform the search from both the directions(start to end and end to start). As one starts from the start and the other from the end the intersection of both the searches is considered to be a goal state. In bi-directional search one direction will be from start to the end state while the other will be from the end state to the start state. The time complexity of the search is greatly reduced. This can be explained with the number of nodes being explored. As time complexity for the search is generally in terms of branching factor raised to the number of steps to the goal node. In the bi-directional search as we move in both directions, the number of nodes to the goal node will be reduced to somewhere near half. Therefore as it is in the branching factor power. This significantly reduces time and the nodes expand and thereby efficiency. For a unidirectional search the time complexity will be O(bk) where b is the branching factor of the nodes and k is the number of steps from the start state to the goal state. For the Bi-directional search as we discussed it will be O(bk/2). Bi-directional search will be most optimal when both the searches meet at the middle.

This search can be easily interpreted by taking an example of two cars. Suppose there are two cars A and B , and A needs to meet B. In uni-directional B is at rest and A moves towards B. In bi-directional A moves towards B and B moves towards A. So we can interpret that in most of the cases bi-directional searches requires less time to meet than unidirectional searches. We shouldn't come to a conclusion that bi-directional search can be used in all cases and it works better than normal search. Bi-directional search is effective under certain conditions which we will discuss in the later part of the document. There is also a disadvantage of the bidirectional search that we need to know the goal state before starting the search. Bi-directional search is most effective when the branching factor from the start and the end is the same. Talking about the optimality of the bi-directional search it will be optimal if the cost is uniform. Bi-directional is also complete if the cost is uniform.

Manuscript created December 2021; This work was developed by the students of Arizona State University.

II. Technical Approach

In this document we are going to discuss two implementations, MM0 and MM algorithms. MM0 and MM algorithms vary in the type of data structure we are going to discuss. MM0 algorithm uses queue and MM algorithm uses Priority Queue. For each of these implementations we maintain two queues one for the forward directions and one for the backward direction. We also use closed sets and action lists to keep the track of the nodes expanded and the actions taken. Firstly we push the start node into a fringe and goal node into another fringe. Along with the nodes we also send the actions which initially are empty lists.

Now we will loop through the nodes until one of the fringe becomes empty(either the forward fringe or the backward fringe). The node we explore is obtained by popping the queue. At every stage we check if the state we are exploring is already explored in the reverse direction. If yes then we return the actions until then added by reversing the last step. For example we reach a state S' by forward direction and this S' is also present in the closed set of backward direction, then we return the states taken to reach S' and add the reversing direction that first explored a path to S' in reverse direction. If this is not the case and if the state S' is not explored in the forward direction then we loop through each unexplored successor and push them in the forward or backward fringe as per the direction we are exploring the nodes. We also add the actions to reach the successor nodes in the closed set dictionary. The same method is implemented in the backward direction too. Firstly we pop an element, and check if it is the goal node. A node is a goal node in bi-directional search if it is already explored in the reverse direction. If it is then we return the required actions as discussed above. If not we travel through each of its successors and push them into the queue and also add the successor node to the closed set with appropriate actions. This is about the implementation of the MM0 algorithm.

```
Initialize closed sets, action lists, and
PriorityQueue for forward and backward
directions as C_F, C_B, A_F, A_B, P_F, P_B.
P_F <- H(S, P, T)
P_B <- H(G, P, F)
F_F <- push((S, A_F), P_F)
F_B <- push((G, A_B), P_B)
while F_F and F_B not empty:
    l_f, A_F <- pop(F_F)
    if l_f is goal state:
        return actions
    if l_f not in C_F:
        for S_N, A, C in successors(l_f):
            update actions
            if S_N not in C_F:
                P_F <- cost + H
                push((S_N,action), P_F)
                C_F <- action
    l_b, A_B <- pop(F_F)
    if l_b is goal state:
        return actions
    if l_b not in C_B:
        for S_N, A, C in successors(l_b):
            update actions
            if S_N not in C_B:
                P_B <- cost + H
                push((S_N,action), P_B)
                C_B <- action
return empty_set
```

Fig. 1. Pseudo code of MM algorithm

MM algorithm is quite similar to MM0 implementation but there are certain differences from MM algorithm such as using priority queue instead of queue, using heuristics. In MM implementation we use the closed sets and action lists similar to the above implementations. We use priority queues for the fringes(both forward and backward). Along with the start and end states and actions we also push the heuristic values of the states in separate fringes. The queues being priority queues always sort the available nodes based on the priority values. The heuristic we implemented is called bidirectional heuristic. As in bidirectional search we have both start state and the end state, it is important to have a function which returns keeping in mind the direction we are exploring. So the function we implement considers the direction and if it is forward we give the Manhattan distance to the goal node and if it is backward we return distance to the start node. We loop through the nodes till one of the fringes becomes empty. The process we are going to discuss is the same for both forward and backward directions. We pop the fringe and see if the node is the goal node. A node is a goal node when the node is in the closed set of the backward direction. That means if we are exploring in

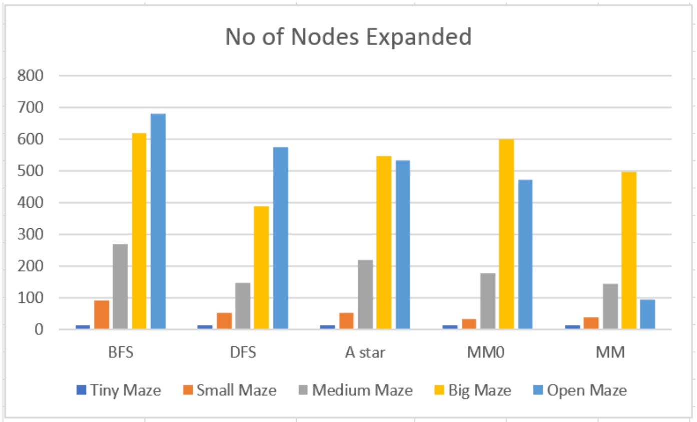

Fig. 2. No of nodes expanded for each maze

the forward direction then we see if the node is in the closed set of the backward direction. If this is not the case then we expand its children as we did in the MM0 implementation. We see if the element is in its own closed set. If not then we get the cost with the help of cost and the heuristic value. We then push it into the fringe and also add the successor to the closed set. The similar procedure is followed for the backward search. The nodes are popped and checked for goal state. If it is not a goal state then the unexplored states of it are passed along with the cost to the fringe which is a priority queue. The successors are also added to the closed set. An important thing to remember is the way we calculate the cost. Cost is an important part in the MM implementation as it is used for prioritizing the nodes to expand. The cost is calculated as the cost till the present node added with the heuristic of the successor node. Now as we discussed both of the implementations it is very important to discuss why we did what we did. Effectively what we are doing is starting the search from both directions instead of one direction. Then we are going node by node based on some rule and checking for goal state. If we think about the two cars example discussed in the abstract, cars meet when car A meets car B or car B meets car A.

The same procedure is followed here. If the node we are exploring in the forward direction is already explored in the backward direction then we are at the point of intersection and therefore we consider it as a goal state. This also applies for the backward search as the meeting is the same whether it is from the starting state or the ending state. Then we need to consider why we are adding elements to the fringe. This is because the present state is not a goal state. We need to explore further states based on the present state and also save the path as we need to backtrack if the path is not saved. The major difference between both the implementations is that we are using heuristics and cost in MM implementation. As we know heuristics help us to reach the goal state faster because of the information it contains about the goal state MM implementation should be faster. Let's see whether this happens in the next section.

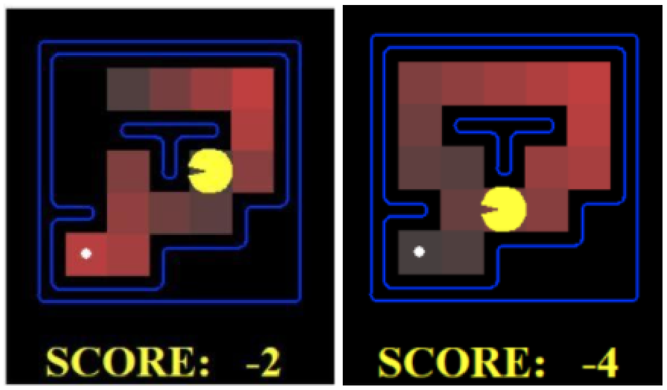

Fig. 3. Tiny Maze Search

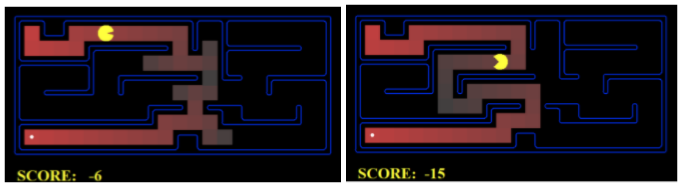

Fig. 4. Medium Maze Search

III. Results,Analyses, and discussion

For Tiny Maze BFS, DFS, UCS takes 15 steps each, while A star, MM0 takes 14 steps each and MM takes just 13 steps. If we observe for a tiny maze the pacman with dfs will follow a different path compared to all other searches. If we run the code for different mazes, we will know why the nodes expanded by the mm0 and mm algorithms are less. Both of these algorithms don't expand the nodes that are to the left as they are not important and with the bidirectional heuristic the pacman knows what all steps to explore.

In the above images the first image is the mm algorithm with bi-directional heuristic whereas the second one is bfs search where all the nodes are expanded in contrast to the few nodes expanded by the mm algorithm. For Small Maze and Small Maze1 MM0 and MM takes less steps compared to others as expected. For both of these MM0 works faster than MM. Below the first image is of MM algorithm and the second one is of MM algorithm. It is also important to observe that in MM algorithm we have expanded only the nodes that we are travelling in while in MM0 algorithm we expanded some other nodes as well. This can be explained by considering heuristics. As MM algorithm considers heuristics it is goal driven and follows the path that takes it near to the goal while the MM0 algorithm which doesn't consider heuristics do the search from both the sides will expand less nodes as it avoids the trap path which increases the nodes expanded.

The same thing also happens for Small Maze as the MM0 algorithm doesn't expand the nodes that appear to be close to the goal. In small maze2 the nodes expanded by the MM algorithm are less than MM0 algorithm because of the initial position of the pacman. The position of the pacman in

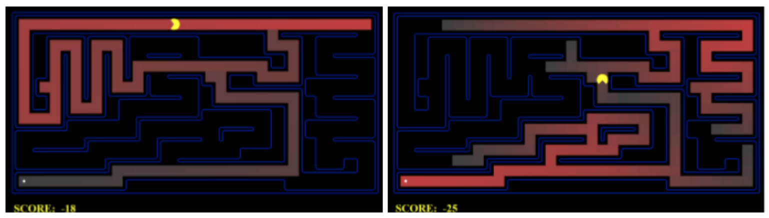

Fig. 5. Big Maze Search

this maze is at the top right. So the long trap path which appears to be close to the goal is avoided. So the MM algorithm only expands the nodes that are important while MM0 algorithm also expands some other nodes. We see an anomaly for the Medium Mazes. We observe that the DFS algorithm outperforms even MM0 and MM algorithms and obviously BFS, UCS. What do you think is the reason for this anomaly? The reason for this anomaly is the nature of DFS. As DFS expands the nodes which are deepest first it expands the nodes to the goal instead of all the nodes as done by other searches. It is the type of maze which decides the number of nodes expanded. As we have seen some mazes are suited for some search. As the mediumMaze needs an algorithm which should expand till the deepest node first DFS suits the most.

In a big maze it is difficult to derive some conclusion as each type of search excels based on the initial position. But we observe that the MM algorithm performs better in all cases. As it is a large maze it may favour search based on the starting position and the path it follows. The important thing to observe is that bi-directional search is better than the normal search. Bi-directional MM0 algorithm worked better than BFS search and MM better than A* search. In open mazes A* search and MM algorithms outperform other searches. MM0 algorithm also performs comparatively well. As it is an open maze there will be very little blocks. So a heuristic helps the search to go in a proper direction and travel less nodes. The idea in an open maze is to get the right direction to travel in. As heuristics gives this both of the heuristic searches perform better. Contours mazes are the type of mazes in which there are no obstacles and it is better if we know to travel in the right direction. We see that MM algorithm differs from other algorithms greatly. This is because of the right direction expansion and travelling in both directions. In other searches almost all the nodes are expanded and thereby the total nodes expanded increases. In open maze and contour maze direction plays an important role. We also observe a pattern followed in both forward and backward directions near the edges. At the edges we observe that some nodes are expanded and from then we get the proper direction. We travel near the edges till we find a path. In custom1 we have a straight and simple path. So BFS, UCS expands all the nodes before reaching the goal state as they go layer by layer. The DFS expands less nodes as all we need is to expand the deepest node first. Next MM and A* algorithm works better because of the directional approach which is expected. In custom2 the obstacles make almost all the searches similar. But in MM algorithm if we see the nodes

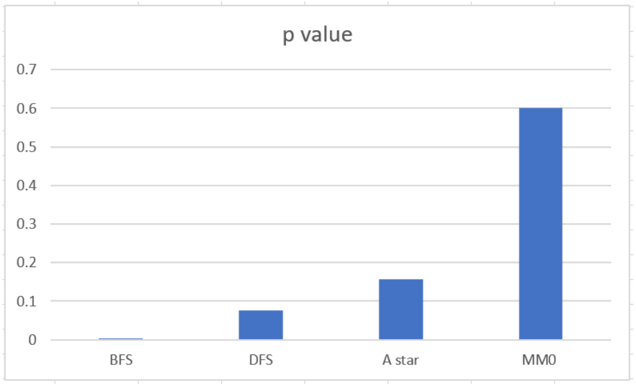

Fig. 6. p Values

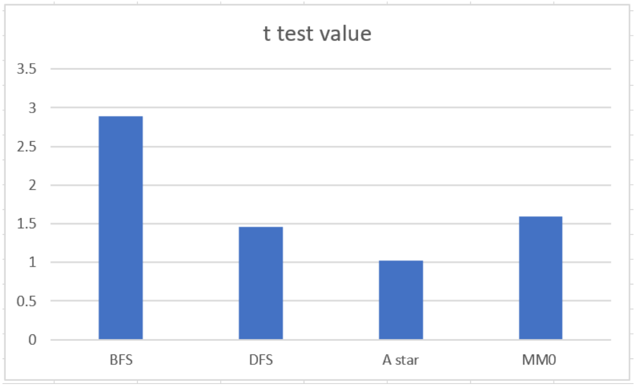

Fig. 7. t test Values

expanded the upper nodes that are near obstacles and above the goal level.

Now as we are done with the node analysis, let's do some statistical analysis. If we consider the mean of various searches, MM algorithm has lesser mean followed by A* and then MM0, DFS. Values are less varied in A* graph followed by MM and MM0 algorithms. Then we calculated t-values of each search with the MM search algorithm to see how close and significant they are. The image which shows the t values and p values are shown. If we set the confidence as 95 percent then we can say that BFS and UCS differ with confidence.

IV. Conclusions, and discussions

To conclude bi-directional search is simply search from both directions which reduces considerable time and nodes expanded, thereby is highly efficient. As discussed for this search the start and the goal state needs to be properly defined and the branching factors must be the same from both ends. This are the optimal conditions under which bi-directional search will be highly useful. A drawback of bidirectional search is it needs more space compared to the uni-directional searches. This is because we will be dealing with both searches i.e forward and backward. The MM and MM0 algorithms are different implementations of bi-directional search. MM search is bi-directional search on A* and MM0 is bi-directional on BFS. As we have seen the results, the bi-directional searches are comparatively better than other searches on various mazes. The implementation of these algorithms is quite simple and similar to other searches and differs only in the way we define the goal state in each direction. The t-tests indicate what type of searches are similar and what are different with certain confidence. Recently comparisons are being made between bi-directional search and solving vertex cover, specialized heuristics for MM implementation, different versions of bi-directional search and so on. Bi-directional search may not be implemented in all the cases and there are space-time tradeoffs, but in some cases bi-directional search can provide a solution that any other search may not provide. There are versions of bi-directional search which are guaranteed to meet in the middle and can beat algorithms in time complexity.

Reference: [1] Holte, Felner, A., Sharon, G., Sturtevant, N. R., Chen, J. (2017). MM: A bidirectional search algorithm that is guaranteed to meet in the middle. Artificial Intelligence, 252, 232–266. https://doi.org/10.1016/j.artint.2017.05.004