# Artificial Intelligence Computer Assignment #1 Report

Hossein Entezari Zarch SID : 810196419
March 2, 2020

**Abstract**

Here we have problem that, includes a table given as input, and we are to apply BFS, IDS and two heuristic A* algorithms to solve this search problem.

## 1 Problem Model

- In order to solve the problem we model it to a search algorithm which we describe more below:
- Initial State: The initial state includes the location of the patients, ambulance and hospitals with their capacity on the map.
- Goal State: The goal state or the state which we are searching for, is a state which its number of patients are zero and all of them are arrived to the hospitals.
- Action: On every state, possible actions is a subset of possible movements with the length of one on the table to neighbour cells for the ambulance, also problem constraints such as walls and patients their self as barrier are considered. So on each state we look at every neighbour cell of the ambulance of the map, if it is empty or containing a hospital then, moving ambulance to the cell is a valid action and, if it is containing a wall then, ambulance can not move to that cell and the movement would not happen, otherwise if the cell contains a patient and it is possible for the patient to move on this direction so, both the ambulance and patient are moved and this would our action on this step.
- Transition Model: on each state when we do an action, we move to the new action and it would be our new state, including its own patients, ambulance and hospitals on the map.
- Path Cost: The path cost between any two neighbour state, which we arrive with a step from first state to the other is considered to be one.

## 2 BFS

- Implementation: In this algorithm we have a queue named frontier, we initialized it with the initial state at first then, while the length of frontier is more than zero, on each iteration, after pop() from the frontier, we expand the popped node and for its children nodes, after checking if they have not been seen already and they are not present in frontier, they would be pushed() to the frontier.
- Double States: In order to manage expanding visited states, we have a ”set” named visited which contains hashed states, so with checking if the hashed state of child is present in visited we would not push the child in frontier, also we check if the child is present in frontier we would not add it to frontier.
- Analysis: The algorithm is complete and finds a way if there exist, also it is optimal in the cases where the cost of all paths are uniformly the same, finally, the time and space complexity of execution the algorithm is $O(b^d)$ where ”b” is branch-factor and ”d” is depth of the target point. In this algorithm we are forced to store the visited state of the cells otherwise our algorithm would not work properly and get stuck in infinite loop.
- Specs: BFS algorithm searches the state space in low depth to high depth points one by one, this would gives us the optimal solution, as an uninformed search algorithm but, if the goal state is far from the initial state, it would need a complex computation to overcome the problem, and finally as a crucial point, it is optimal just, on search spaces with unique path cost, in other words, BFS calculates the path with the fewest steps possible, not with the lowest cost.

# 3 IDS

- Implementation: As a subroutine, we defined a restricted version of DFS which searches for goal state starting from initial state until a special depth far from it, so it would see all the nodes that have a path from initial state to goal state with a length lower than or equal to the highest depth passed to the function. So with calling this subroutine in a loop starting from zero and increasing this variable as depth at every iteration until the DFS function finds the goal state, would make it possible for the IDS algorithm to find the shortest path from initial state to the goal state.
- Double States: In order to handle double states in an efficient way we utilized a few techniques, at first the IDS function have a visited list which passes to the DFS algorithm, so the DFS algorithm on expanding each node, it would be able to find out if the expanding node have been seen by IDS algorithm previously and count it as a double state, and also update this visited list on each expand if needed. Also in each DFS function call as we aim to visit all nodes that have a path lower or equal than given depth, we define and update a dictionary mapping hash of states to the states, which help us on adding new nodes to the stack of the algorithm, that if the node have been added to stack previously (maybe now it is not present in the stack or maybe it is present) with a higher path, so the stored depth in the dictionary would be updated with the new depth and the node would be added to the stack otherwise it would not be added to the stack, hence, thorough this approach, we would be able to perform a more efficient search for the requested nodes.
- Analysis: The algorithm is complete and finds a way if there exist, also it is optimal in the state space with the unique path costs, according to its time execution complexity, it costs O($b^d$) and about the memory consumption it is O(bd), so it outputs the true result with a higher level of complexity compared to BFS, as it sees near nodes to initial point for multiple times, but it needs much lower amount of memory to be executed.

# 4 Uninformed VS Informed Search Algorithms

- We saw the ”BFS” and ”IDS” search algorithms as two uninformed search algorithms, which on the process of search does not have any information about the whole state space and does not any sense when it is far from the solution or it is just near it. but on the other hand, the informed algorithms are available have some approximate measure in themselves to find out what state or path is better to find the goal sooner. On the forward path we have used A* search algorithm, with two special heuristic algorithms approximating the path each state has to arrive to the goal state.

# 5 A*

- Implementation: In the implementation, we have min heap containing a tuple from f (f = heuristic + cost) and the node, this data structure stores the frontier for us, which gives us the node minimum value of ”f” in O(log n), so after that, in a while until the length of the frontier is zero, on each iteration, we pop() a node from the frontier to expand and after adding it’s hash to the visited set, and checking if it is the goal state, we check for its neighbour states, if they are not visited yet, they would be added to the frontier in a tuple with their ”f” value.
- Double States: One part of the double states are handled through visited set, and if a node’s hash is present in it, then it would not be added to the frontier, also in the cases where the child is present in the frontier currently, this would be checked by frontierSet set containing hash of frontier elements, if the node, has a cost higher than, the node which is equal to it and in the frontier we would not add the node to the frontier, but if the node has a lower cost, then the node in the frontier which is accessed in O(1) by froniterSet would changed to invalid by changing its father value to -2, and then the new node, would be added, to the frontier, so after pop() from the frontier each time, we check if the popped node’s father is -2, so we would not do anything on it, through these approaches, we have vanished a lot of duplicate states.
- Analysis: Due to the techniques used in the algorithm and they expressed widely above, the algorithm with an admissible heuristic function would be, optimal, The time complexity of the algorithm mostly depends on the heuristic function we use but, in the worst case it is exponential just like Dijkstra, also according to the

memory complexity, it is exponential.

- Heuristic Function: We know that, heuristic function plays a role to estimate the path cost a state have to arriving the goal state, and in the A* search algorithm, all the time, the node with the less amount of cost and estimated path value, would be chosen from the frontier to expand.
  - Function #1: In this method of heuristic, we have a light function which gives the sum of the amount of patients and capacity of the hospitals as the estimated value for each state, although, we know that, this method is not much smart, but it helps us much in converging to the goal state sooner. This method is not admissible, consider a state having a hospital with capacity 1 and a patient close to it in the neighbour cell, so the optimum path cost to the goal state is 1, but the algorithm gives us 2, as estimated path cost, so all of the time the estimated value is not less than the real value, but most of the time, it is less than the real value.
  - Function #2: In this method of heuristic, through a loop over all of the patients on the map, we get a sum over their least Euclidic Distance to the nearest hospital, so through this approach we have a estimation over how much distance should be passed at least to converge to the goal state, this function is admissible, as it is sum over Euclidic Distance for every patient to the nearest hospital, but we know that, absolutely the real path would be more than, Euclidic Distance, as the constraints of the movement we have in the problem and most of the time maybe, it would not be possible for every patient to go to the nearest hospital cause of capacity they have. So this method always estimates a cost lower or equal than the real, so it is admissible.

# 6 Results

| Algoritm | Test # | Solution Distance | Visited States | Unique States | Execution Time |
|---|---|---|---|---|---|
| BFS | Test #1 | 11 | 845 | 396 | 0.097s |
| BFS | Test #2 | 27 | 20022 | 9227 | 1.129s |
| BFS | Test #3 | 39 | 65739 | 28585 | 5.068s |
| IDS | Test #1 | 11 | 1801 | 451 | 0.198s |
| IDS | Test #2 | 27 | 170159 | 9697 | 16.777s |
| IDS | Test #3 | 39 | 692679 | 30312 | 1m54.422s |
| A* #1 | Test #1 | 11 | 445 | 209 | 0.085s |
| A* #1 | Test #2 | 27 | 14038 | 6515 | 1.072s |
| A* #1 | Test #3 | 39 | 44881 | 19726 | 4.675s |
| A* #2 | Test #1 | 11 | 336 | 158 | 0.083s |
| A* #2 | Test #2 | 27 | 9868 | 4663 | 1.214s |
| A* #2 | Test #3 | 39 | 28069 | 12474 | 4.556s |