Artificial Intelligence Laboratory

Task 5

A* Algorithm

Kunal Kumar -170010012
Aditya Jha -170010011

---

1. Brief description about the domain:

Our Task is based on Rugby Games. We have two teams denoted by '0' and '+' as players in test cases. Our task is to reach the goal with a ball based on heuristic function. We also need to save the ball from the opposing team. Any opponent can take the ball if Ball passes from his neighbour's state.

1. State space:

We have 4 types of state. Same team members are denoted by '0' in input matrix. We are calling these node friend node. We have to pass the ball to our team player. Opponents are denoted by '+' in the input matrix. We are calling opponents player enemy node. We have target state denoted by '*' in matrix. Rest of the free space in the input matrix is free space where players can move.

2. Start node and goal node:

We are starting the game from the corner of Ground. Here we have a 2-D matrix , so we are starting from position (0,0) in matrix.
Our target is to reach the Goal Post which we have denoted by '*' in the input matrix. We can vary this goal state by changing position of * in the input file. Our program will stop once we reach the goal state if the path exists Or terminate when open becomes empty.

3. MOVEGEN and GOALTEST algorithm:

Our Movegen Function is based on the path between players.
i.e. if we can directly reach teammates without crossing any players, then those player's state is the output of our movegen function. We used *Breadth first search* Algorithm with some modification to get neighbouring state.

```
def MoveGen(kk):
    for i in range(len(temp_arr)):
        for j in range(len(temp_arr[0])):
            array[i][j].dis = -1    # reinitialising
distance
    queue = list()      # finding childs
    kk.dis = 0
    neighbour = list()
    queue.append(kk)
    while queue:
        temp = queue.pop(0)
        adjacent = moveable_path(temp)#getting
neighbours
        adjacent = [node for node in adjacent if
node.dis == -1]# removing already visited node
        for node in adjacent:
            node.dis = temp.dis + 1    # updating
distance
            if node.value == 0 or node.value == 3:
                neighbour.append(node)
            else:
                queue.append(node)
    return neighbour
```

As we have already marked our goal node by '*' we have an exact location of the goal state. So we are just comparing coordinates of every node with goal node. If they matched, we are returning *True*.

```
 # checking goal
if node.x == goal_x and node.y== goal_y:
      return True  # goal_x and goal_y are the
coordinates of goal state.
```

2. Heuristic functions considered:

We used three different heuristic functions.

1. Our third heuristic function is sum of area of matrix(length*breadth), Euclidean distance square , Manhattan distance and max(row distance, col distance). Adding Area guarantee the overestimate nature of heuristic function, as it covers every row and every column of matrix as its path.

```
area = row*col     # Area of matrix
   for node in friends:
        xx = node.x
        yy = node.y  # coordinates of node
        node.h = area + (goal_x-xx)**2+(goal_y-yy)**2 +
abs(goal_x-xx)+abs(goal_y-yy) + max(abs(goal_x-xx),
abs(goal_y-yy))
```

2. Our second heuristic function is based on Manhattan distance. We calculated the distance of all friends nodes from the goal state. This function underestimates the cost to the goal state. As It will take two line paths which will always be less than or equal to other paths (we can't cross enemy player).

```
for node in friends:  # list of all friend node
        xx = node.x  # coordinates of node
        yy = node.y
        node.h = abs(goal_x-xx)+abs(goal_y-yy)  # updating value
```

3. Our third heuristic function is based on Euclidean distance. We calculated the distance of all friend nodes from the goal state. This heuristic function satisfies Monotonic property. From triangle property the difference of any two sides of a triangle is less than its third side. Here the third side is the euclidean distance between two nodes and it is always shorter than any path.

```
for node in friends:     # list of all friend node
        xx = node.x   # coordinates of node
        yy = node.y
        euclidean_distance = ((goal_x-xx)**2 +
(goal_y-yy)**2)**(0.5)
        node.h = euclidean_distance   # updating value
```

3. A* algorithm analysis and observation:

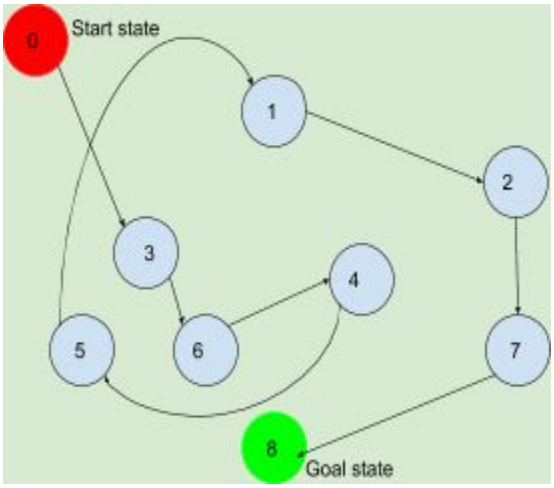
The A* Algorithm combines the best features of Branch & Bound, Dijkstra's algorithm and Best first Search Algorithms. Best first search only looks ahead from the node , seeking a quick path to the goal, while B&B only looks behind, keeping track of the best paths found so far. Algorithm A* does both. A* achieves better performance by using heuristic to guide its search.One major practical drawback is its space complexity (O(b^d)), as it stores all generated nodes in memory. A* Search Algorithms are often used to find the shortest path from one point to another point. A* start from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n)+h(n)$$

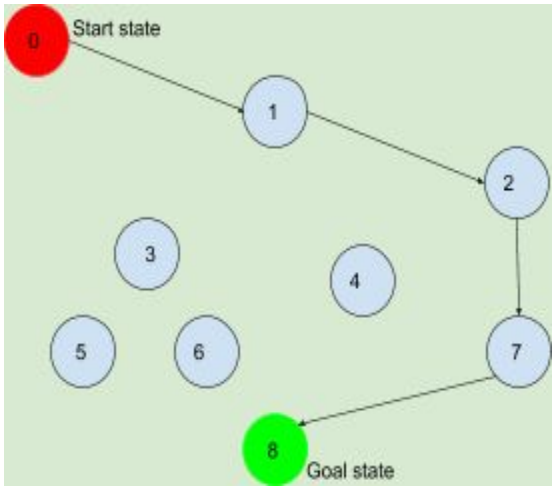
where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.At each step of the algorithm, the node with the lowest $f(n)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

We found that if a path exist from start node to goal node , A* always find the path. A* find the least cost path to the goal. A* is complete and admissible. Both space and time complexity directly depend on heuristic function. If the heuristic h satisfies the additional condition $h(x) \le d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone.. With a monotone heuristic, A* is guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) − h(x)$.

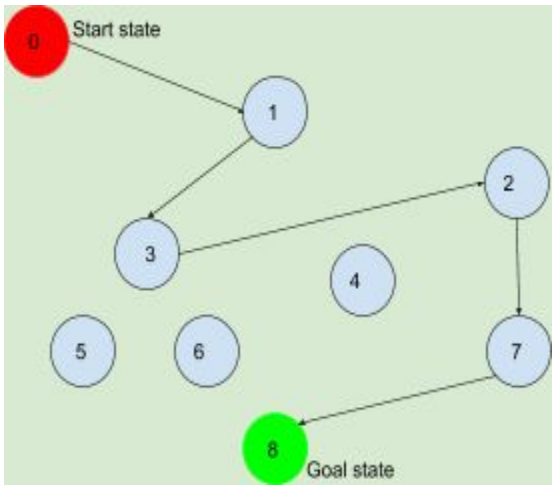
Heuristic 1 → overestimate

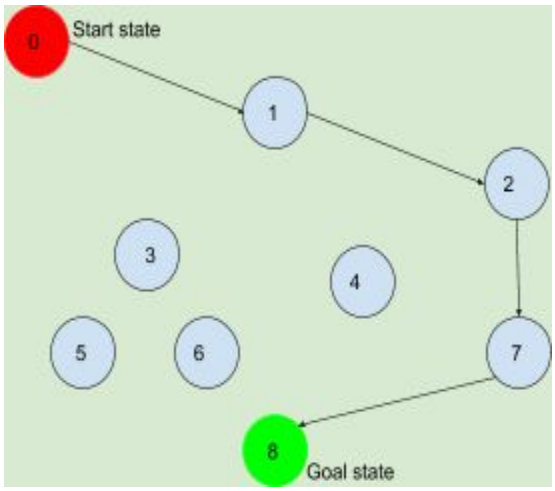


(input3) Optimal path ↓

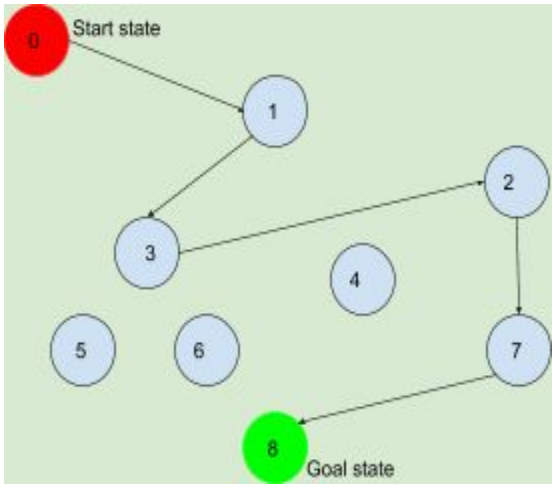


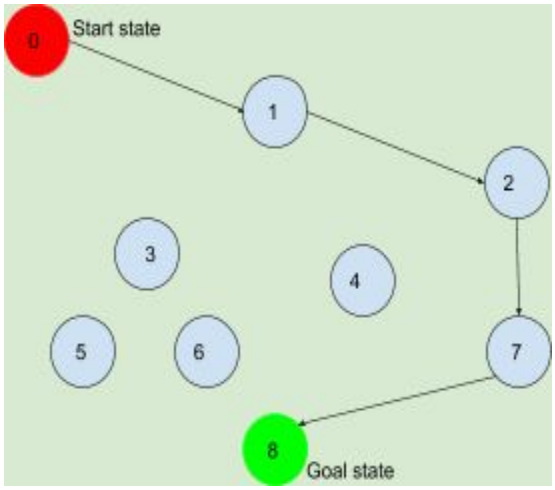
Heuristic 2 → underestimate





Heuristic 3 → monotone property





first→ path visited
second→ optimal path

Heuristic → Input File ↓	Overestimate	Underestimate	Monotone property
input.txt (start→ 0) (Goal→ 4)	0→ 1→ 2→ 4	0→ 2→ 5→ 3→ 4	0→ 2→ 5→ 3→ 4
	0→ 2→ 4 cost=10	0→ 2→ 4 cost=10	0→ 2→ 4 cost=10
Input2.txt (start→ 0) (Goal→ 5)	0→ 3→ 5	0→ 3→ 1→ 2→ 7→ 6→ 5	0→ 1→ 3→ 2→ 7→4 6→ 5
	0→ 3→ 5 cost=42	0→ 1→ 5 cost=36	0→ 1→ 5 cost=36
Input3.txt (start→ 0) (Goal→ 8)	0→ 3→ 6→ 4→ 5→ 1→ 2→ 7→ 8	0→ 1→ 3→ 2 → 7→ 8	0→ 1→ 3→ 2 → 7→ 8
	0→ 1→ 2→ 7→ 8 cost=40	0→ 1→ 2→ 7→ 8 cost=40	0→ 1→ 2→ 7→ 8 cost=40

Thus we can observe that A* with heuristic with monotone and underestimating nature always gives optimal path with least cost. While heuristic with overestimating nature does not always give optimal path with least cost.