Lab1

Introduction to AI, M. MATEI
ISEP - December 2023
by Theophile Wemaere

The A* search algorithm

Question 1. Definitions and properties

1. Give the definition of the optimality of an algorithm

An algorithm is said to be optimal if it finds the solution with the lowest cost or the best possible value for the objective function. For a search algorithm, that would be finding the lowest cost of the path from start to goal among all paths availables.

2. Give the definition of the completeness of an algorithm

An algorithm is said to be complete if it is guaranteed to find a solution if one exists. In other words, a complete algorithm will not get stuck or give up if there is a solution to the problem.

3. Give the Difference between the A* algorithm and the Uniform-cost Search algorithm

The A* algorithm and the Uniform-cost Search algorithm both work by expanding nodes in the graph one at a time until they reach the goal state.

The main difference between the two algorithms is that the A *algorithm uses a heuristic function to guide its search. The heuristic function is an estimate of the distance between the current node and the goal state. This allows the A* algorithm to focus on the most promising paths, which can make it more efficient than Uniform-cost Search.

The A* algorithm is a more powerful and versatile algorithm than Uniform-cost Search. However, Uniform-cost Search is simpler to implement and can be more efficient if the heuristic function is not very accurate.

Question 2. Graph Search

The following pseudo-code is an example of the A* algorithm

```
def a_star(start, goal, graph):
    // Initialize the open and closed sets
    open_set = set([start])
    closed_set = set()
```

```
    // Initialize the gScore and fScore dictionaries
    g_score = {}
    g_score[start] = 0
    f_score = {}
    f_score[start] = g_score[start] + h(start) // h() is the heuristic
function

    // Initialize the came_from dictionary
    came_from = {}
    came_from[start] = None

    // While the open set is not empty
    while open_set:
        // Find the node with the lowest fScore in the open set
        current = None
        for node in open_set:
            if current is None or f_score[node] < f_score[current]:
                current = node

        // If the current node is the goal, reconstruct the path and
return
        if current == goal:
            reconstruct_path(came_from, current)
            return

        // Remove the current node from the open set and add it to the
closed set
        open_set.remove(current)
        closed_set.add(current)

        // For each neighbor of the current node
        for neighbor, weight in graph[current]:
            // If the neighbor is in the closed set, skip it
            if neighbor in closed_set:
                continue

            // Calculate the tentative gScore
            tentative_g_score = g_score[current] + weight

            // If the neighbor is not in the open set or the tentative
gScore is lower than the current gScore, update the gScore and fScore
            if neighbor not in open_set or tentative_g_score <
g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score
```

Question 3. The implementation of the A* research method

The following python script is an example of the A* algorithm implementation :

```
#!/usr/bin/python3

def h(x):
    H = {
    "A":2,
    "B":5,
    "C":2,
    "D":1
    }
    return H[x] if x not in ("Start","Goal") else 0

def a_star(start, stop, graph):
        # FRINGE are nodes spawned but not inspected and CLOSED are nodes
inspected
        # we use set instead of list for easier access
        FRINGE = set([start])
        CLOSED = set([])

        # distances from start to each node
        g = {}
        g[start] = 0

        # previous contains all nodes before each nodes (like a path)
        previous = {}
        previous[start] = start

        while len(FRINGE) > 0:
            # current is the node we are examining in this iteration
            current = None

            # find lowest value of f() for each node
            for candidat in FRINGE:
                if current == None or g[candidat] + h(candidat) <
g[current] + h(current):
                    current = candidat;

            # is it the goal node ?
            if current == stop:
                path = []
                while previous[current] != current:
                    path.append(current)
                    current = previous[current]
                path.append(start)
                path.reverse()
                print(' -> '.join(path))
                return 0
```

```
            # get all neighbors of the current node
            for node_name,weight in graph[current]: # return list of
["node_name",weight]
                # if this is a new node ?
                if node_name not in FRINGE and node_name not in CLOSED:
                    FRINGE.add(node_name)
                    previous[node_name] = current
                    g[node_name] = g[current] + weight

                else:
                    # check if it is worth it to visit the current
neighbor (node_name)
                    if g[node_name] > g[current] + weight:
                        g[node_name] = g[current] + weight
                        previous[node_name] = current

                        if node_name in CLOSED:
                            CLOSED.remove(node_name)
                            FRINGE.add(node_name)

            # all neighbors have been inspected, remove it
            FRINGE.remove(current)
            CLOSED.add(current)

        print('Error, no path found')
        return -1

if __name__ == "__main__":
    try:
        graph = {
            "Start":[("A",2),("B",3)],
            "A":[("Start",2),("C",4)],
            "B":[("Start",3),("D",4)],
            "C":[("A",4),("D",1),("Goal",2)],
            "D":[("Start",5),("B",4),("C",1),("Goal",5)],
            "Goal":[("C",2),("D",5)]
        }
        a_star('Start', 'Goal',graph)
    except KeyboardInterrupt:
        print("\nExiting...")
        exit(1)
```