AU 332 Artificial Intelligence: Principles and Techniques

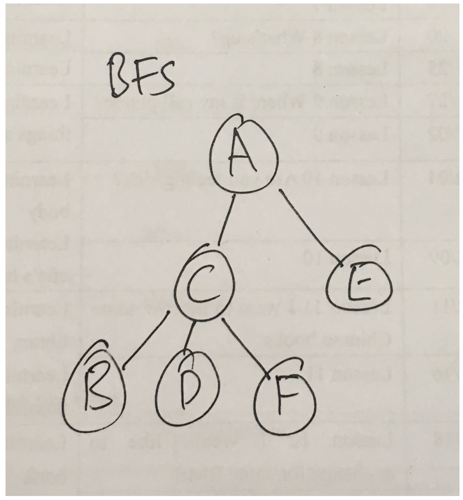
By: Harry Lording (717030990012)

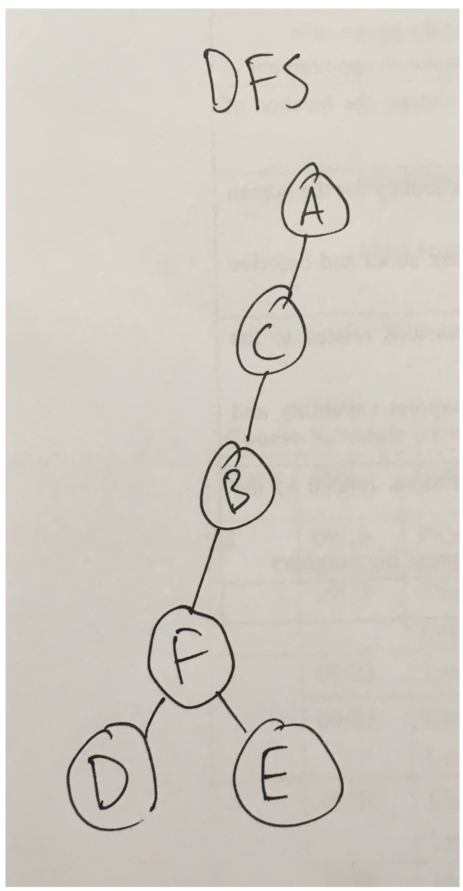
HW#: 1

September 23, 2019

I. QUESTION 1

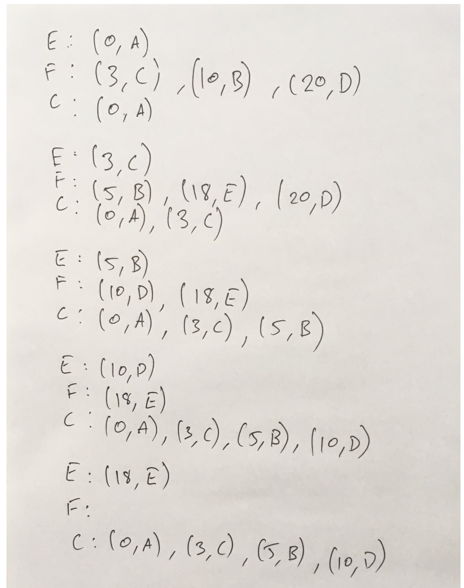
In the first picture we see the expanded tree if breath first search was executed on the graph given in the question. The second picture we see the result if depth first search was executed.





II. QUESTION 2

In the picture below the expanded node, fringe list and closed list can be seen if uniform cost search was executed on the graph in question. The shortest path from the start node A to the goal node E can be observed to be A→C→E.



E : (0, A)
F : (3, C) , (10, B) , (20, D)
C : (0, A)

E : (3, C)
F : (5, B) , (18, E) , (20, D)
C : (0, A), (3, C)

E : (5, B)
F : (10, D), (18, E)
C : (0, A), (3, C), (5, B)

E : (10, D)
F : (18, E)
C : (0, A), (3, C), (5, B), (10, D)

E : (18, E)
F :
C : (0, A), (3, C), (5, B), (10, D)

III. QUESTION 3

The aim of this question was to write out the fringe and closed list for an A* search algorithm for the given 7x5 grid in the question. The grid had a wall in the middle that the algorithm had to navigate around.

For this question I used Manhattan distance as my heuristic due to its simplicity and because it is admissible. In cases of ties, the node with the highest x coordinate would be chosen first, and in the case of a tie again, preference was given to the node with the highest y value.

The key for the list is (g(n) + h(n), ’x,y’). Closed list is for the iteration is followed by the fringe (the first row in the table).

Fringe and Closed Lists
(0+4, ’1,2’)
(1+3, ’2,2’), (1+5, ’1,3’), (1+5, ’1,1’), (1+6, ’0,2’)
(1,2)
(2+4, ’2,3’), (2+4, ’2,1’), (1+5, ’1,3’), (1+5, ’1,1’), (1+5, ’0,2’)
(1,2), (2,2)
(2+4, ’2,1’), (1+5, ’1,3’), (1+5, ’1,1’), (1+5, ’0,2’), (3+5,’2,4’)
(1,2), (2,2), (2,3)
(1+5, ’1,3’), (1+5, ’1,1’), (1+5, ’0,2’), (3+5, ’2,4’), (3+5, ’2,0’)
(1,2), (2,2), (2,3), (2,1)
(1+5, ’1,1’), (1+5, ’0,2’), (3+5, ’2,4’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’)
(1,2), (2,2), (2,3), (2,1), (1,3)
(1+5, ’0,2’), (3+5, ’2,4’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1)
(3+5, ’2,4’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2)
(4+4, ’3,4’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4)
(5+3, ’4,4’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4), (3,4)
(6+2, ’5,4’), (6+2, ’4,3’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4), (3,4), (4,4)
(7+1, ’5,3’), (6+2, ’4,3’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1), (7+3, ’6,4’)
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4), (3,4), (4,4), (5,4)
(8+0, ’5,2’), (6+2, ’4,3’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1), (7+3, ’6,4’), (8+2, ’6,3’),
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4), (3,4), (4,4), (5,4), (5,3)
(6+2, ’4,3’), (3+5, ’2,0’), (2+6, ’1,4’), (2+6, ’0,3’), (2+6, ’1,0’), (2+6, ’0,1), (7+3, ’6,4’), (8+2, ’6,3’),
(1,2), (2,2), (2,3), (2,1), (1,3), (1,1), (0,2), (2,4), (3,4), (4,4), (5,4), (5,3), (5,2)

IV. QUESTION 4

BFSvsDFS.py

```
def reconstruct_path(came_from, start, goal):
    """
    Given a dictionary of came_from where its key is the node
    character and its value is the parent node, the start node
    and the goal node, compute the path from start to the end

    Arguments:
    came_from -- a dictionary indicating for each node as the key and
                 value is its parent node
    start -- A character indicating the start node
    goal --  A character indicating the goal node

    Return:
    path. -- A list storing the path from start to goal. Please check
             the order of the path should from the start node to the
             goal node
    """

    #print ("EXECUTING RECONSTRUCT")

    path = []
    visited = set()
    ### START CODE HERE ### ( 6 line of code)
    currNode = goal
    path.append(goal)
    while 1:
        currNode = came_from.get(currNode)
        if currNode in visited:
            return "You're in a loop! Cannot find start node. Exiting"
        else:
            path.append(currNode)
            visited.add(currNode)
        if currNode == start:
            break
    ### END CODE HERE ###
    return list(reversed(path))
    # return path

def breadth_first_search(graph, start, goal):
    """
    Given a graph, a start node and a goal node
    Utilize breadth first search algorithm by finding the path from
    start node to the goal node
    Use early stoping in your code
    This function returns back a dictionary storing the information of each node
    and its corresponding parent node
    Arguments:
    graph -- A dictionary storing the edge information from one node to a list
```

```
            of other nodes
    start -- A character indicating the start node
    goal --  A character indicating the goal node

    Return:
    came_from -- a dictionary indicating for each node as the key and
                 value is its parent node
    """

    #print ("EXECUTING BFS")

    came_from = {}
    came_from[start] = None
    # initialize a queue and place the start node in the queue
    q = Queue()
    q.put(start)
    visited = set()
    #print("start is " + start + " goal is " + goal )
    ### START CODE HERE ### ( 10 line of code)
    ### while the queue isn't empty connected nodes from the node popped from the queue
    while q.qsize() != 0:

        currNode = q.get()
        #print("currNode is " + currNode)
        if currNode not in visited:
            ### add node popped to the visited list
            visited.add(currNode)
            neighbours = graph.neighbors(currNode)
            for neighbour in neighbours:
                #print(neighbour)
                ### queue neighbour
                q.put(neighbour)
                if neighbour not in visited:
                    came_from[neighbour] =  currNode
                ### if the edge is the goal we've reached the goal and we return came_from
                if neighbour == goal:
                    return came_from
        #print("\n")

    ### END CODE HERE ###
    return came_from

def depth_first_search(graph, start, goal):
    """
    Given a graph, a start node and a goal node
    Utilize depth first search algorithm by finding the path from
    start node to the goal node
    Use early stoping in your code
    This function returns back a dictionary storing the information of each node
    and its corresponding parent node
    Arguments:
    graph -- A dictionary storing the edge information from one node to a list
             of other nodes
    start -- A character indicating the start node
```

```
goal --  A character indicating the goal node

Return:
came_from -- a dictionary indicating for each node as the key and
            value is its parent node
"""
#print ("EXECUTING DFS")

came_from = {}
came_from[start] = None
# initialize a queue and place the start node in the queue
s = LifoQueue()
s.put(start)
visited = set()
#print("start is " + start + " goal is " + goal )
### START CODE HERE ### ( 10 line of code)
### while the queue isn't empty connected nodes from the node popped from the queue
while s.qsize() != 0:

    currNode = s.get()
    #print("currNode is " + currNode)
    if currNode not in visited:
        ### add node popped to the visited list
        visited.add(currNode)
        neighbours = graph.neighbors(currNode)
        for neighbour in neighbours:
            #print(neighbour)
            ### queue neighbour
            s.put(neighbour)
            if neighbour not in visited:
                came_from[neighbour] =  currNode
            ### if the edge is the goal we've reached the goal and we return came_from
            if neighbour == goal:
                return came_from
    #print("\n")

### END CODE HERE ###
return came_from
```

UniformCostSearch.py

```
def reconstruct_path(came_from, start, goal):
    """
    Given a dictionary of came_from where its key is the node
    character and its value is the parent node, the start node
    and the goal node, compute the path from start to the end

    Arguments:
    came_from -- a dictionary indicating for each node as the key and
                 value is its parent node
    start -- A character indicating the start node
    goal --  A character indicating the goal node

    Return:
    path. -- A list storing the path from start to goal. Please check
             the order of the path should from the start node to the
             goal node
    """
    path = []
    visited = set()
    ### START CODE HERE ### ( 6 line of code)
    currNode = goal
    path.append(goal)
    while 1:
        currNode = came_from.get(currNode)
        if currNode in visited:
            return "You're in a loop! Cannot find start node. Exiting"
        else:
            path.append(currNode)
            visited.add(currNode)
        if currNode == start:
            break
    ### END CODE HERE ###
    return list(reversed(path))

    ### END CODE HERE ###
    # return path


def uniform_cost_search(graph, start, goal):
    """
    Given a graph, a start node and a goal node
    Utilize uniform cost search algorithm by finding the path from
    start node to the goal node
    Use early stoping in your code
    This function returns back a dictionary storing the information of each node
    and its corresponding parent node
    Arguments:
    graph -- A dictionary storing the edge information from one node to a list
             of other nodes
    start -- A character indicating the start node
```

```
goal --  A character indicating the goal node

Return:
came_from -- a dictionary indicating for each node as the key and
            value is its parent node
"""
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

visited = set()
q = PriorityQueue()
q.put((0, start))

while q.qsize() != 0:
    currNode = q.get()
    #print("currNode is " + currNode[1] + " and the cost is " + str(currNode[0]))
    currNode = currNode[1]
    if currNode not in visited:

        visited.add(currNode)
        if currNode == goal:
            return came_from, cost_so_far

        neighbours = graph.neighbors(currNode)
        for neighbour in neighbours:
            #print(neighbour)
            if neighbour not in came_from.keys():
                q.put((cost_so_far[currNode] +  graph.get_cost(currNode, neighbour), neighbour))
                came_from[neighbour] =  currNode
                cost_so_far[neighbour] = cost_so_far[currNode] +  graph.get_cost(currNode, neighbou
            elif cost_so_far[currNode] +  graph.get_cost(currNode, neighbour) < cost_so_far[neighbo
                q.put((cost_so_far[currNode] +  graph.get_cost(currNode, neighbour), neighbour))
                came_from[neighbour] =  currNode
                cost_so_far[neighbour] = cost_so_far[currNode] +  graph.get_cost(currNode, neighbou
            else:
                continue
    #print("\n")

### END CODE HERE ###
return came_from, cost_so_far
```

AStarSearch.py

```
def heuristic(graph, current_node, goal_node):
    """
    Given a graph, a start node and a next nodee
    returns the heuristic value for going from current node to goal node
    Arguments:
    graph -- A dictionary storing the edge information from one node to a list
             of other nodes
    current_node -- A character indicating the current node
    goal_node --  A character indicating the goal node

    Return:
    heuristic_value of going from current node to goal node
    """
    ### START CODE HERE ### ( 15 line of code)
    current_node_location = graph.locations[current_node]
    x1 = current_node_location[0]
    y1 = current_node_location[1]

    goal_node_location = graph.locations[goal_node]
    x2 = goal_node_location[0]
    y2 = goal_node_location[1]

    ### END CODE HERE ###
    return sqrt( ((x1-x2) ** 2) + ((y1-y2) ** 2) )

def A_star_search(graph, start, goal):
    """
    Given a graph, a start node and a goal node
    Utilize A* search algorithm by finding the path from
    start node to the goal node
    Use early stoping in your code
    This function returns back a dictionary storing the information of each node
    and its corresponding parent node
    Arguments:
    graph -- A dictionary storing the edge information from one node to a list
             of other nodes
    start -- A character indicating the start node
    goal --  A character indicating the goal node

    Return:
    came_from -- a dictionary indicating for each node as the key and
                value is its parent node
    """

    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    visited = set()
```

```
q = PriorityQueue()
q.put((0, start))

### START CODE HERE ### ( 15 line of code)
while q.qsize() != 0:
    currNode = q.get()
    #print("currNode is " + currNode[1] + " and the cost is " + str(currNode[0]))
    currNode = currNode[1]
    if currNode not in visited:

        visited.add(currNode)
        if currNode == goal:
            return came_from, cost_so_far

        neighbours = graph.neighbors(currNode)
        for neighbour in neighbours:
            #print(neighbour)
            # if it is the first instance of visiting the node add to dictionaries
            if neighbour not in came_from.keys():
                q.put((cost_so_far[currNode] +  graph.get_cost(currNode, neighbour) + heuristic(gra
                came_from[neighbour] =  currNode
                cost_so_far[neighbour] = cost_so_far[currNode] +  graph.get_cost(currNode, neighbou
            # else compare it to the cost of the previous path
            elif cost_so_far[currNode] +  graph.get_cost(currNode, neighbour) < cost_so_far[neighbo
                q.put((cost_so_far[currNode] +  graph.get_cost(currNode, neighbour) + heuristic(gra
                came_from[neighbour] =  currNode
                cost_so_far[neighbour] = cost_so_far[currNode] +  graph.get_cost(currNode, neighbou
            else:
                continue
    #print("\n")

### END CODE HERE ###
return came_from, cost_so_far
```

V. QUESTION 5

For the programming assignment part, extra credits will be given if you completed the following cases. 1. Check if the goal and start state are valid nodes in the graph, return error handling message. 2. Check whether the graph satisfies the consistency of heuristics