

AU 332 ARTIFICIAL INTELLIGENCE: PRINCIPLES AND TECHNIQUES

By: Yangqian Wu (517021911095)

HW#: 1

September 22, 2019

I. INTRODUCTION

A. Purpose

Homework 1 is about Search Algorithm. This homework guides us through several useful Search Algorithms including the algorithms belows:

- breadth first search
- depth first search
- uniform cost search
- A star search

B. Equipment

There is a minimal amount of equipment to be used in this homework. The few requirements are listed below:

- Python 3.6
- Sublime Text
- Anaconda

II. ASSIGNMENT 1

1 Graph Traversal

In Figure 1, the start node is A. Please draw an expanding tree structure for the graph using depth-first search algorithm and the breadth-first search algorithm. If the graph has n nodes and the maximum degree for each node is d , what is the complexity of BFS and DFS?

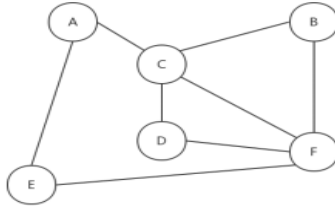
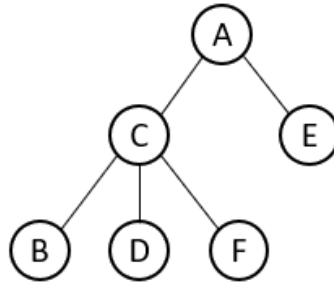


Figure 1: Solve DFS and BFS based on the given graph. Node A is the start node

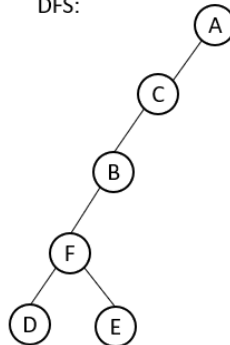
(a) Assignment 1's description

BFS:



(b) BFS's expanding tree

DFS:



(c) DFS's expanding tree

FIG. 1: Assignment 1's answer.

- BFS:the Time Complexity is $O(nd)$;the Space Complexity is $O(n)$.
- DFS:the Time Complexity is $O(nd)$;the Space Complexity is $O(n)$.

III. ASSIGNMENT 2

2 Uniform Cost Search Algorithm

In Figure 2, the start node is A and the goal node is E, calculate the shortest path from node A to E using UCS method. Write step by step update of the fringe list and closed list.

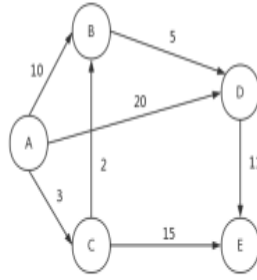


Figure 2: Uniform cost graph search problem

(a) Assignment 2's description

Fringe List	Closed List
A:0	∅
A->C:3 A->B:10 A->D:20	A
A->C->B:5 A->B:10 A->C->E:18 A->D:20	A C
A->C->B->D:10 A->C->E:18 A->D:20	A C B
A->C->E:18 A->C->B->D->E:21	A C B D
∅	A C B D E

(b) UCS's steps

FIG. 2: Assignment 2's answer.

IV. ASSIGNMENT 3

3 A* Algorithm

Write out the complete path finding process (fringe list and close list) from the green grid to the red grid using A* algorithm (blue grids are obstacles). The green grid is the start location and the red grid is the goal location. Write out your choice of h heuristic function. use the row and column number as reference to the location of the cell. The action space are up, down, left and right. The cost for each action is 1.

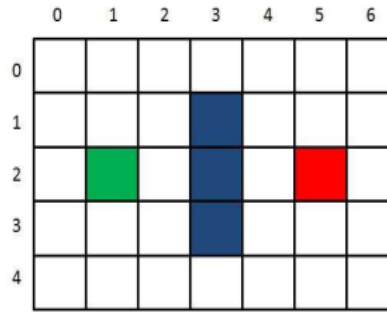


Figure 3: Use A* to solve for the maze

(a) Assignment 3's description

For this question, we can select the Manhattan Distance from the start to the goal as h heuristic function.

As usual, we can select the movement has been cost as g function. Then, we can list the procedure as below:

Fringe List	Closed List	Answer
(1,2):0+4	\emptyset	\emptyset
(2,2):1+3 (1,1):1+5 (1,3):1+5 (0,2):1+5	(1,2)	(1,2)
(2,1):2+4 (2,3):2+4 (1,1):1+5 (1,3):1+5 (0,2):1+5	(1,2) (2,2)	(1,2)->(2,2)
(2,3):2+4 (1,1):1+5 (1,3):1+5 (0,2):1+5 (2,0):3+5 (1,1):3+5	(1,2) (2,2) (2,1)	(1,2)->(2,2)->(2,1)
(1,1):1+5 (1,3):1+5 (0,2):1+5 (2,0):3+5 (1,1):3+5 (2,4):3+5 (1,3):3+5	(1,2) (2,2) (2,1) (2,3)	(1,2)->(2,2)->(2,3)
(1,3):1+5 (0,2):1+5 (2,0):3+5 (2,4):3+5 (1,3):3+5 (1,0):2+6 (0,1):3+5	(1,2) (2,2) (2,1) (2,3) (1,1)	(1,2)->(1,1)
(0,2):1+5 (2,0):3+5 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3)	(1,2)->(1,3)
(2,0):3+5 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2)	(1,2)->(0,2)
(3,0):4+4 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0)	(1,2)->(2,2)->(2,1) ->(2,0)

(b) A star's steps

(4,0):5+3 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0)
(5,0):6+2 (4,1):6+2 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0) (4,0)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0) ->(4,0)
(5,1):7+1 (4,1):6+2 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (6,0):7+3 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0) (4,0) (5,0)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0) ->(4,0)->(5,0)
(5,2):8+0 (4,1):6+2 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (6,0):7+3 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0) (4,0) (5,0) (5,1)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0) ->(4,0)->(5,0) ->(5,1)
(4,1):6+2 (2,4):3+5 (1,0):2+6 (0,1):3+5 (1,4):2+6 (0,4):2+6 (0,1):2+6 (0,3):2+6 (6,0):7+3 (1,0):4+6	(1,2) (2,2) (2,1) (2,3) (1,1) (1,3) (0,2) (2,0) (3,0) (4,0) (5,0) (5,1) (5,2)	(1,2)->(2,2)->(2,1) ->(2,0)->(3,0) ->(4,0)->(5,0) ->(5,1)->(5,2)

(c) A star's steps(next)

FIG. 3: Assignment 3's answer.

V. ASSIGNMENT 4

4 Programming Assignment

Please complete the partial code of *BFSvsDFS.py*, *UniformCostSearch.py* and *AStarSearch.py*.

(a) Assignment 4's description

```
Large graph
came from DFS {'S': None, 'C': 'S', 'A': 'S', 'D': 'A', 'B': 'A', 'H': 'B', 'G': 'H', 'F': 'H', 'E': 'G', 'K': 'E', 'L': 'C', 'J': 'L', 'I': 'L'}
path from DFS ['S', 'A', 'B', 'H', 'G', 'E']
came from BFS {'S': None, 'A': 'S', 'C': 'S', 'B': 'A', 'D': 'A', 'L': 'C', 'H': 'B', 'F': 'D', 'I': 'L', 'J': 'L', 'G': 'H', 'K': 'I', 'E': 'G'}
path from BFS ['S', 'A', 'B', 'H', 'G', 'E']
Small graph
came from DFS {'A': None, 'D': 'A', 'B': 'A', 'C': 'B', 'E': 'D'}
path from DFS ['A', 'D', 'E']
came from BFS {'A': None, 'B': 'A', 'D': 'A', 'C': 'B', 'E': 'D'}
path from BFS ['A', 'D', 'E']
```

(b) BFS and DFS's answer

```
Small graph
came from UCS {'A': None, 'B': 'A', 'D': 'A', 'C': 'B', 'E': 'D'}
cost from UCS {'A': 0, 'B': 2, 'D': 4, 'C': 5, 'E': 7}
path from UCS ['A', 'D', 'E']
Large graph
came from UCS {'S': None, 'A': 'S', 'B': 'S', 'C': 'S', 'D': 'A', 'H': 'B', 'L': 'C', 'F': 'D', 'G': 'H', 'E': 'G'}
cost from UCS {'S': 0, 'A': 7, 'B': 2, 'C': 3, 'D': 11, 'H': 3, 'L': 5, 'F': 16, 'G': 5, 'E': 7}
path from UCS ['S', 'B', 'H', 'G', 'E']
```

(c) UCS's answer


```
Small Graph
came from Astar {'A': None, 'B': 'A', 'D': 'A', 'C': 'B', 'E': 'D'}
cost from Astar {'A': 0, 'B': 2, 'D': 4, 'C': 5, 'E': 7}
path from Astar ['A', 'D', 'E']
Large Graph
came from Astar {'S': None, 'A': 'S', 'B': 'S', 'C': 'S', 'D': 'A', 'H': 'B', 'L': 'C', 'F': 'D', 'G': 'H', 'E': 'G'}
cost from Astar {'S': 0, 'A': 7, 'B': 2, 'C': 3, 'D': 11, 'H': 3, 'L': 5, 'F': 16, 'G': 5, 'E': 7}
path from Astar ['S', 'B', 'H', 'G', 'E']
```

(d) A star's answer

FIG. 4: Assinment 4's answer.

VI. ASSIGNMENT 5

5 Extra credit

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.

(a) Assignment 5's description

```
if start not in graph.edges.keys() or goal not in graph.edges.keys(): #检查start和goal是否为graph里的node
    print("Not valid node!")
    return False
```

(b) 1. Check if the goal and start state are valid nodes in the graph, return error handling message.

```
def Judgeconsistency(graph,goal): #检查启发式函数的一致性
    for node1 in graph.edges:
        for node2 in graph.edges[node1]:
            #一旦存在两个节点不满足该关系式,就返回False
            if heuristic(graph,node1,goal) > heuristic(graph,node2,goal) + graph.get_cost(node1,node2):
                return False
    return True #否则返回True
```

(c) Check whether the graph satisfies the consistency of heuristics.

FIG. 5: Assignment 5's answer.

VII. HOMEWORK'S CODE

This section will consist of the final code of each homework.

A. code 1:BFSvsDFS

```
1  -*- coding: utf-8 -*-
2  from queue import LifoQueue
3  from queue import Queue
4  from queue import PriorityQueue
5
6  class Graph:
7      """
8      Defines a graph with edges, each edge is treated as dictionary
9      look up function neighbors pass in an id and returns a list of
10     neighboring node
11     """
12
13     def __init__(self):
14         self.edges = {}
15
16     def neighbors(self, id):
17         # check if the edge is in the edge dictionary
18         if id in self.edges:
19             return self.edges[id]
20         else:
21             print("The node", id, "is not in the graph")
22             return False
23
24     def reconstruct_path(came_from, start, goal):
25         """
26         Given a dictionary of came_from where its key is the node
27         character and its value is the parent node, the start node
28         and the goal node, compute the path from start to the end
29
30         Arguments:
31         came_from -- a dictionary indicating for each node as the key and
32         value is its parent node
33         start -- A character indicating the start node
34         goal -- A character indicating the goal node
35
36         Return:
37         path -- A list storing the path from start to goal. Please check
38         the order of the path should from the start node to the
39         goal node
40         """
41         path = []
42         ### START CODE HERE ### ( 6 line of code)
43         path.append(goal)
44         while(goal != start):
45             goal = came_from[goal]
46             path.append(goal)
47         path.reverse()
48         ### END CODE HERE ###
49         return path
50
51     def breadth_first_search(graph, start, goal):
52         """
53         Given a graph, a start node and a goal node
54         Utilize breadth first search algorithm by finding the path from
55         start node to the goal node
56         Use early stopping in your code
57         This function returns back a dictionary storing the information of each node
58         and its corresponding parent node
59
60         Arguments:
61         graph -- A dictionary storing the edge information from one node to a list
62         of other nodes
63         start -- A character indicating the start node
64         goal -- A character indicating the goal node
65
66         Return:
67         came_from -- a dictionary indicating for each node as the key and
68         value is its parent node
69         """
70         came_from = {}
71         came_from[start] = None
72         ### START CODE HERE ### ( 10 line of code)
73         if start not in graph.edges.keys() or goal not in graph.edges.keys():
74             print("Not valid node!")
75             return False
76
77         fringe = Queue()
78         closed = []
79
80         fringe.put(start)
81         closed.append(start)
82         while not fringe.empty():
83             current = fringe.get()
```

```

        for child in graph.neighbors(current): #
85         if child not in closed:
87             fringe.put(child)
            closed.append(child)
            came_from[child] = current
89         if child == goal: # goal
            break
91     ### END CODE HERE ###
    return came_from
93
95
97 def depth_first_search(graph, start, goal):
    """
99     Given a graph, a start node and a goal node
    Utilize depth first search algorithm by finding the path from
101     start node to the goal node
    Use early stopping in your code
103     This function returns back a dictionary storing the information of each node
    and its corresponding parent node
105     Arguments:
    graph -- A dictionary storing the edge information from one node to a list
107     of other nodes
    start -- A character indicating the start node
109     goal -- A character indicating the goal node

111     Return:
    came_from -- a dictionary indicating for each node as the key and
113     value is its parent node
    """
115     came_from = {}
    came_from[start] = None
117     ### START CODE HERE ### ( 10 line of code)
    if start not in graph.edges.keys() or goal not in graph.edges.keys(): #
        start goal graph node
119         print("Not valid node!")
        return False

121     fringe = LifoQueue() #Fringe List FILO ( )
123     closed = [] #Closed List

125     fringe.put(start) # start
127     closed.append(start) # start
    while not fringe.empty(): # Fringe
129         List goal
        current = fringe.get()
        for child in graph.neighbors(current): #

            if child not in closed:
                fringe.put(child)
                closed.append(child)
                came_from[child] = current
            if child == goal: # goal
                break
135     ### END CODE HERE ###
    return came_from
137
139
141
143 # The main function will first create the graph, then use depth first search
# and breadth first search which will return the came_from dictionary
# then use the reconstruct path function to rebuild the path.
145 if __name__ == "__main__":
    small_graph = Graph()
    small_graph.edges = {
147         'A': ['B', 'D'],
149         'B': ['A', 'C', 'D'],
        'C': ['A'],
151         'D': ['E', 'A'],
        'E': ['B']
    }
    large_graph = Graph()
    large_graph.edges = {
153         'S': ['A', 'C'],
155         'A': ['S', 'B', 'D'],
157         'B': ['S', 'A', 'D', 'H'],
159         'C': ['S', 'L'],
        'D': ['A', 'B', 'F'],
161         'E': ['G', 'K'],
        'F': ['H', 'D'],
163         'G': ['H', 'E'],
        'H': ['B', 'F', 'G'],
165         'I': ['L', 'J', 'K'],
        'J': ['L', 'I', 'K'],
167         'K': ['I', 'J', 'E'],
        'L': ['C', 'I', 'J']
    }
    print("Large_graph")
    start = 'S'
    goal = 'E'
173     came_fromDFS = depth_first_search(large_graph, start, goal)
    print("came_fromDFS", came_fromDFS)
175     pathDFS = reconstruct_path(came_fromDFS, start, goal)
    print("path_fromDFS", pathDFS)
177     came_fromBFS = breadth_first_search(large_graph, start, goal)
    print("came_fromBFS", came_fromBFS)
179     pathBFS = reconstruct_path(came_fromBFS, start, goal)
    print("path_fromBFS", pathBFS)
181

```

```

183     print("Small_graph")
184     start = 'A'
185     goal = 'E'
186     came_fromDFS = depth_first_search(small_graph, start, goal)
187     print("came_fromDFS", came_fromDFS)
188     pathDFS = reconstruct_path(came_fromDFS, start, goal)
189     print("path_fromDFS", pathDFS)
190     came_fromBFS = breadth_first_search(small_graph, start, goal)
191     print("came_fromBFS", came_fromBFS)
192     pathBFS = reconstruct_path(came_fromBFS, start, goal)
193     print("path_fromBFS", pathBFS)

```

B. code 2:UniformCostSearch

```

# -*- coding: utf-8 -*-
2 from queue import LifoQueue
3 from queue import Queue
4 from queue import PriorityQueue
5
6 class Graph:
7     """
8     Defines a graph with edges, each edge is treated as dictionary
9     look up function neighbors pass in an id and returns a list of
10    neighboring node
11
12    """
13    def __init__(self):
14        self.edges = {}
15        self.edgeWeights = {}
16        self.locations = {}
17
18    def neighbors(self, id):
19        if id in self.edges:
20            return self.edges[id]
21        else:
22            print("The node", id, "is not in the graph")
23            return False
24
25    def get_node_location(self, id):
26        return self.nodeLocation[id]
27
28    def get_cost(self, from_node, to_node):
29        #print("get_cost_for", from_node, to_node)
30        nodeList = self.edges[from_node]
31        #print(nodeList)
32        try:
33            edgeList = self.edgeWeights[from_node]
34            return edgeList[nodeList.index(to_node)]
35        except ValueError:
36            print("From node", from_node, "to", to_node, "does not exist a direct connection")
37            return False
38
39    def reconstruct_path(came_from, start, goal):
40        """
41        Given a dictionary of came from where its key is the node
42        character and its value is the parent node, the start node
43        and the goal node, compute the path from start to the end
44
45        Arguments:
46        came_from -- a dictionary indicating for each node as the key and
47        value is its parent node
48        start -- A character indicating the start node
49        goal -- A character indicating the goal node
50
51        Return:
52        path -- A list storing the path from start to goal. Please check
53        the order of the path should from the start node to the
54        goal node
55        """
56        path = []
57        ### START CODE HERE ### ( 6 line of code)
58        path.append(goal)
59        while(goal != start):
60            goal = came_from[goal]
61            path.append(goal)
62        path.reverse()
63        ### END CODE HERE ###
64        return path
65
66    def uniform_cost_search(graph, start, goal):
67        """
68        Given a graph, a start node and a goal node
69        Utilize uniform cost search algorithm by finding the path from
70        start node to the goal node
71        Use early stopping in your code
72        This function returns back a dictionary storing the information of each node
73        and its corresponding parent node
74
75        Arguments:
76        graph -- A dictionary storing the edge information from one node to a list
77        of other nodes
78        start -- A character indicating the start node
79        goal -- A character indicating the goal node

```

```

80 ----Return:
82 ----came_from---a_dictionary_indicating_for_each_node_as_the_key_and
-----value_is_its_parent_node
84 ----
86     came_from = {}
87     cost_so_far = {}
88     came_from[start] = None
89     cost_so_far[start] = 0
90     ### START CODE HERE ### ( 15 line of code)
91     if start not in graph.edges.keys() or goal not in graph.edges.keys(): #
92         print("Not valid node!")
93         return False
94
95     fringe = PriorityQueue() #Fringe List
96     closed = [] #Closed List
97
98     fringe.put(start, cost_so_far[start]) # start
99     closed.append(start) # start
100     while not fringe.empty(): # Fringe List goal
101         current = fringe.get()
102         if current == goal: # goal goal UCS
103             break
104         for child in graph.neighbors(current): #
105             if child not in closed:
106                 cost_so_far[child] = cost_so_far[current] + graph.get_cost(current, child)
107                 fringe.put(child, cost_so_far[child])
108                 closed.append(child)
109                 came_from[child] = current
110     ### END CODE HERE ###
111     return came_from, cost_so_far
112
113
114
115 # The main function will first create the graph, then use uniform cost search
116 # which will return the came_from dictionary
117 # then use the reconstruct path function to rebuild the path.
118 if __name__ == "__main__":
119     small_graph = Graph()
120     small_graph.edges = {
121         'A': ['B', 'D'],
122         'B': ['A', 'C', 'D'],
123         'C': ['A'],
124         'D': ['E', 'A'],
125         'E': ['B']
126     }
127     small_graph.edgeWeights={
128         'A': [2, 4],
129         'B': [2, 3, 4],
130         'C': [2],
131         'D': [3, 4],
132         'E': [5]
133     }
134
135     large_graph = Graph()
136     large_graph.edges = {
137         'S': ['A', 'B', 'C'],
138         'A': ['S', 'B', 'D'],
139         'B': ['S', 'A', 'D', 'H'],
140         'C': ['S', 'L'],
141         'D': ['A', 'B', 'F'],
142         'E': ['G', 'K'],
143         'F': ['H', 'D'],
144         'G': ['H', 'E'],
145         'H': ['B', 'F', 'G'],
146         'I': ['L', 'J', 'K'],
147         'J': ['L', 'I', 'K'],
148         'K': ['I', 'J', 'E'],
149         'L': ['C', 'I', 'J']
150     }
151     large_graph.edgeWeights = {
152         'S': [7, 2, 3],
153         'A': [7, 3, 4],
154         'B': [2, 3, 4, 1],
155         'C': [3, 2],
156         'D': [4, 4, 5],
157         'E': [2, 5],
158         'F': [3, 5],
159         'G': [2, 2],
160         'H': [1, 3, 2],
161         'I': [4, 6, 4],
162         'J': [4, 6, 4],
163         'K': [4, 4, 5],
164         'L': [2, 4, 4]
165     }
166
167     print("Small_graph")
168     start = 'A'
169     goal = 'E'
170     came_from, UCS = uniform_cost_search(small_graph, start, goal)
171     print("came_from UCS", came_from)
172     print("cost_from UCS", cost_so_far)
173     pathUCS = reconstruct_path(came_from, UCS, start, goal)
174     print("path_from UCS", pathUCS)
175
176     print("Large_graph")
177     start = 'S'
178     goal = 'E'

```

```

180 came_from_UCS, cost_so_far = uniform_cost_search(large_graph, start, goal)
181 print("came_from_UCS", came_from_UCS)
182 print("cost_from_UCS", cost_so_far)
183 path_UCS = reconstruct_path(came_from_UCS, start, goal)
184 print("path_from_UCS", path_UCS)

```

C. code 3:AStarSearch

```

# -*- coding: utf-8 -*-
2 from queue import LifoQueue
  from queue import Queue
4 from queue import PriorityQueue

6 class Graph:
  """
8   ___Defines_a_graph_with_edges,_each_edge_is_treated_as_dictionary
  ___look_up_function_neighbors_pass_in_an_id_and_returns_a_list_of
10  ___neighboring_node

12  ___"""
  def __init__(self):
14      self.edges = {}
15      self.edgeWeights = {}
16      self.locations = {}

18      def neighbors(self, id):
19          if id in self.edges:
20              return self.edges[id]
21          else:
22              print("The_node_", id, "_is_not_in_the_graph")
23              return False

24      # this function get the g(n) the cost of going from from_node to
25      # the to_node
26      def get_cost(self, from_node, to_node):
27          #print("get_cost_for_", from_node, to_node)
28          nodeList = self.edges[from_node]
29          #print(nodeList)
30          try:
31              edgeList = self.edgeWeights[from_node]
32              return edgeList[nodeList.index(to_node)]
33          except ValueError:
34              print("From_node_", from_node, "_to_", to_node, "_does_not_exist_a_direct_connection")
35              return False

36      def reconstruct_path(came_from, start, goal):
37          """
38          ___Given_a_dictionary_of_came_from_where_its_key_is_the_node
39          ___character_and_its_value_is_the_parent_node,_the_start_node
40          ___and_the_goal_node,_compute_the_path_from_start_to_the_end

41          ___Arguments:
42          ___came_from___a_dictionary_indicating_for_each_node_as_the_key_and
43          ___value_is_its_parent_node
44          ___start___A_character_indicating_the_start_node
45          ___goal___A_character_indicating_the_goal_node

46          ___Return:
47          ___path___A_list_storing_the_path_from_start_to_goal._Please_check
48          ___the_order_of_the_path_should_from_the_start_node_to_the
49          ___goal_node

50          """
51          path = []
52          ### START CODE HERE ### ( 6 line of code)
53          path.append(goal)
54          while(goal != start):
55              goal = came_from[goal]
56              path.append(goal)
57          path.reverse()
58          ### END CODE HERE ###
59          return path

60      def heuristic(graph, current_node, goal_node):
61          """
62          ___Given_a_graph,_a_start_node_and_a_next_node
63          ___returns_the_heuristic_value_for_going_from_current_node_to_goal_node

64          ___Arguments:
65          ___graph___A_dictionary_storing_the_edge_information_from_one_node_to_a_list
66          ___of_other_nodes
67          ___current_node___A_character_indicating_the_current_node
68          ___goal_node___A_character_indicating_the_goal_node

69          ___Return:
70          ___heuristic_value_of_going_from_current_node_to_goal_node

71          """
72          heuristic_value = 0
73          ### START CODE HERE ### ( 15 line of code)
74          heuristic_value = abs(graph.locations[current_node][0] - graph.locations[goal_node][0])\
75              + abs(graph.locations[current_node][1] - graph.locations[goal_node][1])
76          #

```

```

84     ### END CODE HERE ###
85     return heuristic_value
86
87 def A_star_search(graph, start, goal):
88     """
89     Given a graph, a start node and a goal node
90     Utilize A* search algorithm by finding the path from
91     start node to the goal node
92     Use early stopping in your code
93     This function returns back a dictionary storing the information of each node
94     and its corresponding parent node
95     Arguments:
96     graph -- A dictionary storing the edge information from one node to a list
97             of other nodes
98     start -- A character indicating the start node
99     goal -- A character indicating the goal node
100
101     Return:
102     came_from -- a dictionary indicating for each node as the key and
103                value is its parent node
104     """
105
106     came_from = {}
107     cost_so_far = {}
108     came_from[start] = None
109     cost_so_far[start] = 0
110
111     ### START CODE HERE ### ( 15 line of code)
112     if start not in graph.edges.keys() or goal not in graph.edges.keys(): #
113         start, goal, graph, node
114         print("Not valid node!")
115         return False
116
117     fringe = PriorityQueue() #Fringe List
118     closed = [] #Closed List
119
120     fringe.put(start, cost_so_far[start] + heuristic(graph, start, goal)) #
121     closed.append(start) # start
122     while not fringe.empty(): # Fringe List goal
123         current = fringe.get()
124         if current == goal: # goal goal A
125             break
126         for child in graph.neighbors(current): #
127             if child not in closed:
128                 cost_so_far[child] = cost_so_far[current] + graph.get_cost(current, child)
129                 fringe.put(child, cost_so_far[child] + heuristic(graph, child, goal))
130                 closed.append(child)
131                 came_from[child] = current
132     ### END CODE HERE ###
133     return came_from, cost_so_far
134
135 def Judgeconsistency(graph, goal): #
136     for node1 in graph.edges:
137         for node2 in graph.edges[node1]:
138             # False
139             if heuristic(graph, node1, goal) > heuristic(graph, node2, goal) + graph.get_cost(node1, node2):
140                 return False # True
141
142 # The main function will first create the graph, then use A* search
143 # which will return the came_from dictionary
144 # then use the reconstruct path function to rebuild the path.
145 if __name__ == "__main__":
146     small_graph = Graph()
147     small_graph.edges = {
148         'A': ['B', 'D'],
149         'B': ['A', 'C', 'D'],
150         'C': ['A'],
151         'D': ['E', 'A'],
152         'E': ['B']
153     }
154     small_graph.edgeWeights={
155         'A': [2, 4],
156         'B': [2, 3, 4],
157         'C': [2],
158         'D': [3, 4],
159         'E': [5]
160     }
161     small_graph.locations={
162         'A': [4, 4],
163         'B': [2, 4],
164         'C': [0, 0],
165         'D': [6, 2],
166         'E': [8, 0]
167     }
168
169     large_graph = Graph()
170     large_graph.edges = {
171         'S': ['A', 'B', 'C'],
172         'A': ['S', 'B', 'D'],
173         'B': ['S', 'A', 'D', 'H'],
174         'C': ['S', 'L'],
175         'D': ['A', 'B', 'F'],
176         'E': ['G', 'K'],
177         'F': ['H', 'D'],
178         'G': ['H', 'E'],
179         'H': ['B', 'F', 'G'],
180         'I': ['L', 'J', 'K'],
181         'J': ['L', 'I', 'K'],

```



```

182     'K': ['I', 'J', 'E'],
183     'L': ['C', 'I', 'J']
184 }
185 large_graph.edgeWeights = {
186     'S': [7, 2, 3],
187     'A': [7, 3, 4],
188     'B': [2, 3, 4, 1],
189     'C': [3, 2],
190     'D': [4, 4, 5],
191     'E': [2, 5],
192     'F': [3, 5],
193     'G': [2, 2],
194     'H': [1, 3, 2],
195     'I': [4, 6, 4],
196     'J': [4, 6, 4],
197     'K': [4, 4, 5],
198     'L': [2, 4, 4]
199 }
200 large_graph.locations = {
201     'S': [0, 0],
202     'A': [-2, -2],
203     'B': [1, -2],
204     'C': [6, 0],
205     'D': [0, -4],
206     'E': [6, -8],
207     'F': [1, -7],
208     'G': [3, -7],
209     'H': [2, -5],
210     'I': [4, -4],
211     'J': [8, -4],
212     'K': [6, -7],
213     'L': [7, -3]
214 }
215 print("Small_Graph")
216 start = 'A'
217 goal = 'E'
218 came_from_Astar, cost_so_far = A_star_search(small_graph, start, goal)
219 print("came_from_Astar", came_from_Astar)
220 print("cost_from_Astar", cost_so_far)
221 pathAstar = reconstruct_path(came_from_Astar, start, goal)
222 print("path_from_Astar", pathAstar)
223
224
225 print("Large_Graph")
226 start = 'S'
227 goal = 'E'
228 came_from_Astar, cost_so_far = A_star_search(large_graph, start, goal)
229 print("came_from_Astar", came_from_Astar)
230 print("cost_from_Astar", cost_so_far)
231 pathAstar = reconstruct_path(came_from_Astar, start, goal)
232 print("path_from_Astar", pathAstar)

```

VIII. DISCUSSION & CONCLUSION

The goal of this homework was to revise and get familiar with Search Algorithm. Different from BFS and DFS, UCS use weight to determine the priority of each code, moreover, A star is more like a combination of UCS and Greedy. I will revise it and keep learning along the way.

Thanks for teacher and assistant's help.

Fighting!